Multi-User Virtual Reality

by

Jonathan Pugh

B.Sc. Hons (University of Tasmania)

Submitted in fulfilment

of the requirements for the degree of

Master of Science

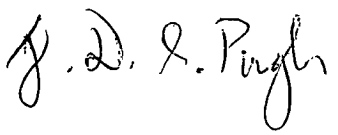
University of Tasmania

Hobart

January 2001

School of Computing

This Thesis contains no material that has been accepted for a degree or diploma by the University or any other institution, except by way of background information and duly acknowledged in the Thesis, and to the best of my knowledge and belief no material previously published or written by any other person except where due acknowledgement is made in the text of the Thesis.



Jonathan David Stewart Pugh (mailto:jdspugh@email.com)

Abstract

Due to the extensive use of modem technology and the advent of mobile communications, bandwidth usage by Internet applications such as multi-user virtual reality systems needs to be kept to a minimum. In traditional multi-user virtual reality simulations, synchronization data about deterministic objects must be regularly sent to remote simulations in order to maintain consistency among the simulations. Using these systems, a maximum of about *ten* objects can be simultaneously synchronized over a typical modem connection.

There is scope for eliminating synchronization information caused by deterministic objects by taking advantage of their predictability. Using the efficient rolling mechanism provided by the Replicated Time Warp mechanism (presented in this thesis), the need to send synchronization data for deterministic objects is eliminated at the expense of extra local computation required to predict the deterministic objects.

By eliminating the bandwidth restrictions, the Replicated Time Warp system enables virtual worlds far more complex than previously possible to be created and synchronized over modem or wireless connections. Virtual Worlds containing *one hundred or more* dynamic objects can be simultaneously synchronized over a modem connection, depending on the speed of the local computer.

Acknowledgments

Thanks to Charles Lakos, my supervisor, for his direction and willingness to take me aboard with this different research topic. Thanks to Merridy Pugh and Gordan Wallace for their thorough proofreading. Thanks to my parents for supporting me, no matter where my interests may lie.

Contents

1 INTRODUCTION

1.1 Rationale

Virtual worlds are real-time simulations that give the user a strong sense of immersion within the simulated world (usually done primarily through the use of 3D graphics). Since these virtual worlds often imitate the real world, the experience is termed 'Virtual Reality' (VR). Multi-user virtual reality enables people to interact closely with each other within these virtual worlds through the use of computer networks, regardless of the participants' actual geographical locations.

Ever since the concept of virtual worlds has existed, one of the main aims has been to enable multi-user participation (Macedonia et al 1994). As an example, at VR World 95 several speakers said that they would not consider a single-user environment as true virtual reality. Dave Sims writes in his review about VR World 95 that 'other users' is clearly the hot new element in VR (Sims 1995).

Applications of multi-user virtual reality are almost limitless and include simulation environments such as group military training and vehicle or weapon prototyping (see SIMNET, DIS & NPSNET), cooperative work environments, group learning environments and group entertainment environments.

Multi-user virtual reality systems first became popularised in military circles (see SIMNET), but the technology developed there has been spreading through the academic and commercial worlds. It is filtering down through these systems and will soon become an accepted part of everyday computing activities on the Internet. The current commercial driving force in multi-user virtual reality is locally networked (Ethernet LANs) or Internet networked (online) gaming systems. This push towards multi-user virtual reality has been helped by a recent boom in 3D technologies for personal computers. Graphics previously possible only on high-end workstations worth thousands of dollars is now available on personal computers for only a few hundred dollars through the use of specialized 3D accelerator cards.

Large companies such as Intel are moving in this direction with new multi-media technologies such as Intel's new MMX Pentium chips, which enable faster and higher quality 3D graphics as well as other multi-media enhancements. Major software companies embracing 3D technologies include Microsoft with their Direct3D technology and Apple with their Quickdraw3D technology.

There is also a lot of interest in the recent VRML (Virtual Reality Modelling Language) standards that have become popular on the Internet. VRML has been specifically

designed for use over the Internet and currently has two versions: VRML 1 and VRML 2. VRML 1 is simply a file format for 3D data, whereas VRML 2 supports animation of the 3D data as well. There is a lot of interest in producing multi-user VRML, with plenty of discussion occurring on relevant VRML mailing lists and newsgroups.

As an example of the enthusiasm in multi-user virtual reality, a proposal for multi-user VRML, 'Living Worlds', is supported by Apple, Fujitsu, IBM, Intel, Silicon Graphics, Sony and many other companies (Honda et al 1997). Its first draft was frozen at VRML97 and the second draft is under development. When a standard for multi-user VRML is agreed upon, it will be a major boost to multi-user virtual reality on the Internet, just as the previous VRML standard has made single-user Virtual Reality popular on the Internet.

Several small companies are now emerging that are embracing the technology used in large-scale multi-user military simulations and are adapting it to online games. MaK Technologies is making the most of both worlds by developing a training system for the US Marines that is planned to be marketed both as a simulation system for the military as well as a networked multi-player game (*The Australian* April 29 1997). It is only a matter of time before more socially useful applications become commercially viable. A few simple multi-user virtual reality systems are currently available for use on the Internet, which have chat facilities as well as limited interaction (Trueman 1995).

In order for the vast majority of the population to use multi-user virtual reality technology, any system that is developed must be able to run on the Internet using current personal computer technology. The majority of Internet access is still via modem, and so the system should, if possible, be able to accommodate these users as well (see Figure 1). Aiming at this audience imposes bandwidth constraints as well as introducing large latencies that need to be dealt with. These limitations also apply to more recent 'wireless' communications technologies, which are becoming increasingly popular (Heim 1997).

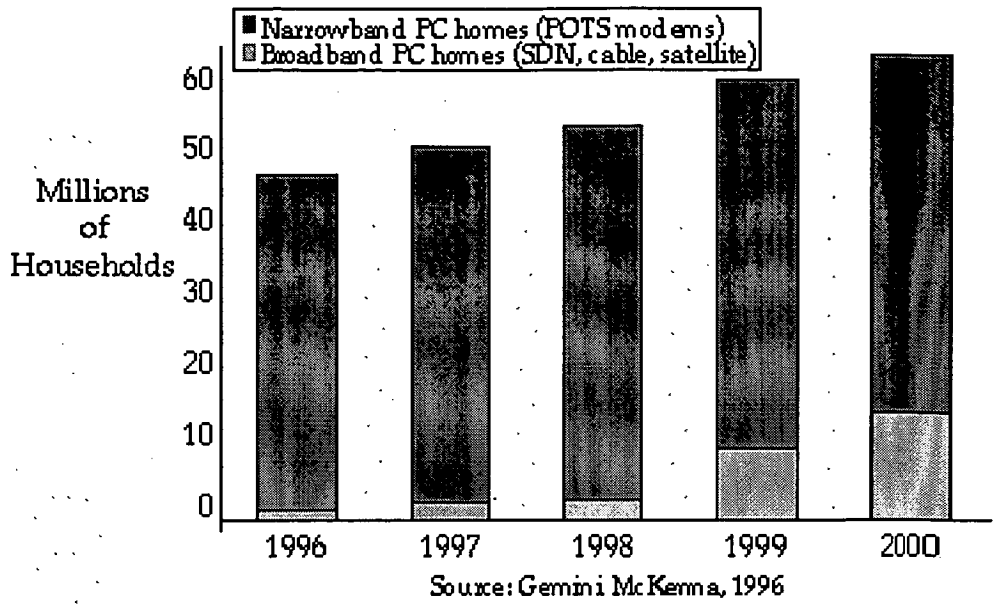


Figure 1 Estimated Modem Usage Statistics (Intel 1997)

One must also realize that any system should attempt to reduce bandwidth as much as possible to allow many multi-user virtual worlds to be run simultaneously over the Internet without clogging up the network. Currently military simulations enable thousands of simultaneous users, but are run over their own dedicated networks.

It is for the above reasons I have decided to tackle multi-user virtual reality with an aim of producing a multi-user virtual reality system that has minimal bandwidth requirements, can accommodate high latency connections and will run on current personal computers.

1.2 Requirements for Multi-User Virtual Worlds

Each multi-user virtual reality system has a minimum set of requirements to meet. These requirements vary from system to system, but various common elements can be found. The most general of these requirements include *consistency, scalability* and *responsiveness.*

These particular areas are important for all multi-user virtual reality systems and are the focus of most multi-user virtual reality studies. Other requirements may be present, depending on the application at hand. Examples of extra requirements include security requirements, fault tolerance requirements, support for spoken interaction or comprehensive run-time modification and extension.

Due to the vast array of applications for multi-user virtual reality, the final degrees of consistency, scalability and responsiveness required will be determined by the application at hand.

Scalability

The system must be able to support the required number of participants and shared objects for the purposes of the particular application. Numbers of users and shared objects range from tens for collaborative work applications, to hundreds of thousands for military simulations. The software architecture is considered scalable if its responsiveness and consistency does not degrade excessively as the number of users increases.

Responsiveness

Studies have shown that delays of more than 100ms in a user interface begin to be perceived by humans and adversely affect their performance (Bailey 1982, Woodson 1987, Fluckiger 1995). The total elapsed time between a user action and the sensory feedback should generally be less than 100ms. Multi-user virtual reality systems should thus ideally attempt to keep user interface delays below 100ms. The degree to which humans are affected by delays is application dependent. Tasks requiring speed and accuracy are affected the most. The system must try to support the responsiveness requirements of the application at hand.

Improving a multi-user virtual reality system's user interface delays involves shortening both response times and notification times. The *response time* is the time it takes a user's interface to reflect its own actions (Ellis et al 1991). The *notification time* is the time required for a user's action to be propagated to the other users' interfaces (Ellis et al 1991). The minimum notification time possible is the same as the response time plus the time it takes for a message to traverse from the sender to the receiver (end-to-end delay). The optimal response time of a multi-user application involves having the local changes reflected as fast as the same application running as a single user application (Ellis et al 1991, Karsenty & Beaudouin-Lafon 1993).

Consistency

Within a multi-user virtual reality system, each user should see consistent views of the parts of the virtual worlds they are interested in. The notion of consistency enables collaboration. The degree of consistency maintained within various multi-user systems varies. Certain applications allow less consistency, which simplifies their design, but also reduces the degree of collaboration that can be obtained. Designing virtual worlds that do not have consistency requirements is trivial, because they are no longer multi-user.

A measurable quantity that relates to consistency is the time is takes for any event to be committed. The commit time is the time it takes for a local user's action to be reflected locally and to be guaranteed never to be altered. The *optimal commit* time is the same time it would take a single user application to process the event. Because

alteration of the user's actions is disconcerting to the user, the commit time should also generally attempt to stay below the 100ms threshold.

1.3 Aims

The aim of this research is to produce a software architecture for the development of multi-user virtual worlds targeted at the largest home market—modem users. The system must meet the requirements of scalability, responsiveness and consistency as needed by these users. In particular, the system should reduce the required bandwidth and network end-to-end delay to their minimum possible values; bandwidth and end-to-end delay being the problem areas for modem users.

For simplicity, only the 3D graphical animation components of multi-user virtual worlds are considered. Sound and video, for example, are not considered.

1.4 Definitions

The following definitions are given in a multi-user virtual reality context:

- *Behaviour* — a set of instructions that determine how an object in a virtual world acts.
- *Commit Time* — the time it takes for a local user's action to be reflected locally and to be guaranteed never to be altered.
- *Concurrency* — when more than one process accesses an object at a time.
- *Concurrency Control* — a process that maintains consistency between virtual worlds.
- *Entity* — an *object* in a virtual world that is usually rendered.
- *Host* — a computer connected to a network.
- *NAK* — negative acknowledgement (see Reliable Multicasting).
- *Network Access Delay* — the time necessary at the source to wait for the medium to be available, or for the network to be ready to accept the block of information (Fluckiger 1995).
- *Network Access Speed* — the frequency at which bits may be sent or received during transmission periods at the interface between the end-system and the network (Fluckiger 1995).
- *Network Bit Transmission Delay* — the time necessary to transmit the sequence of bits of a block once the network is ready. For a given block size, this delay only depends on the access speed (Fluckiger 1995).

- *Network End-to-End Delay* — this measure also takes into account the time it takes for the data to be transmitted onto the medium and received from the medium, not just the network transit time. It is equal to *Access Delay + Network Latency + Bit Transmission Delay.*
- *Network Latency* — the time it takes for an empty message to travel across a network from the source application layer to the destination application layer.
- *Network Transit Delay* — same as *Network Latency.*
- *Notification Time* — the time required for a user's action to be propagated to every user's interface (Ellis et al 1991)
- *Object* — an object in a virtual world usually has a unique identifier, data associated with it representing its state and a view for rendering. Object may also have *behaviour.*
- *Optimistic Concurrency Control* — more than one user can modify an object at a time; inconsistencies are resolved later.
- *Peer* — a host that acts both as a client and as a server.
- *Peer-to-Peer* — hosts can send messages directly to each other and do not rely on a central server.
- *Pessimistic Concurrency Control* — only one user can modify an object at a time. Inconsistencies are thus guaranteed not to occur.
- *Response Time* — the time it takes for a user's interface to reflect their own actions (Ellis et al 1991)
- *Responsiveness* — a combined measure proportional to both response and notification times.
- *TCP* — Transmission Control Protocol. Used for sending messages reliably across the Internet.
- *Totally-Ordered Multicast* — when a sequence of messages are transmitted to a group by totally-ordered multicast the messages reach all of the members of the group in the same order.
- *UDP* — User Datagram Protocol. Used for sending messages unreliably across the Internet.
- *Virtual Reality (VR)* — an alternate reality created by a *Virtual World.*
- *Virtual Reality System* — a system designed to simulate *Virtual Worlds.*
- *Virtual World* — a real-time simulation that gives the user a strong sense of immersion within the simulated world.

1.5 Originality

The following are the main contributions of this thesis:

- An analysis of multi-user virtual reality techniques from various multi-user virtual reality systems, and grouping of these techniques into general categories. These general categories are then used to re-analyse various multi-user virtual reality systems.
- The adaptation of a Replicated Time Warp mechanism for real-time multi-user interaction by synchronizing the Time Warp replicas with the local real-time clocks and synchronizing the remote local real-time clocks with each other. Also by providing a mechanism to handle interactive input and output through a reversible execution concurrency control mechanism.
- The application of Time Warping to the Replicated Time Warp system to reduce the sizes of rollbacks, while still obtaining the benefits of smoothing the perceived effects of end-to-end delay that the Time Warping provides.

1.6 Chapter Overview

Chapter 1, Introduction provides the aims of this study and defines some requirements for multi-user virtual reality systems.

Chapter 2, Design Issues provides background information necessary to analyse and design multi-user virtual worlds. Several multi-user virtual reality techniques are described and it is shown how each technique can be used to help meet aspects of the requirements of multi-user virtual worlds. This information will be used to compare various multi-user virtual reality systems in the Related Work and Replicated Time Warp chapters.

Chapter 3, Related Work describes various multi-user virtual reality systems that exist today. Each system will be related to the requirements and techniques for meeting these requirements as described in the previous chapter.

Chapter 4, Replicated Time Warp presents the background of the Replicated Time Warp system and explains how it works. It analyses and discusses its performance, and compares it to methods described in Chapter 2.

Chapter 5, Conclusion shows how the Replicated Time Warp mechanism meets the aims and requirements presented in the first chapter. It provides a summary of the advantages and disadvantages of the Replicated Time Warp and suggests some future work.

Figure 2 shows a flow diagram of how the chapters relate to each other.

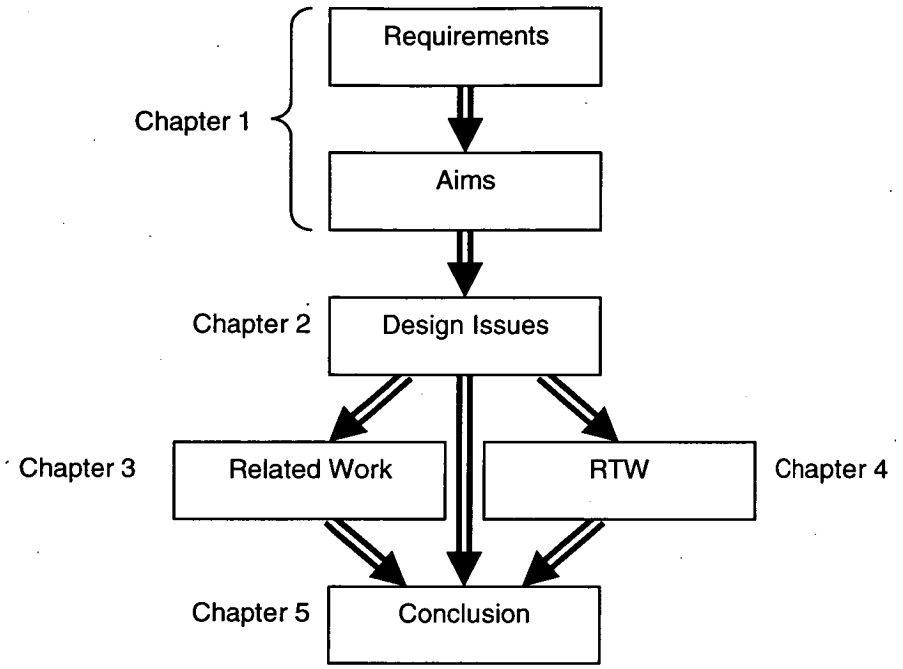


Figure 2 Chapter Linkage

2 DESIGN ISSUES

In order to meet the requirements for multi-user virtual worlds set out in the Introduction chapter, the software that distributes the virtual world must be designed carefully. This chapter provides background information necessary to design or analyse multi-user virtual reality software architectures. Several techniques used in multi-user virtual reality systems are described and how they can help meet requirements for multi-user virtual reality systems.

2.1 Message Passing

There are generally three ways of passing messages on a network: *unicasting*, *multicasting* and *broadcasting*. These methods can send messages either reliably (the message is guaranteed to reach its destination) or unreliably (the message is not guaranteed to reach its destination). The choice of unicasting, multicasting or broadcasting has an impact on the response times and scalability of a multi-user virtual reality system. These methods and their implications are discussed in the following sections.

Unicasting

Unicasting is the process of sending a single message from one host to one other host on a network (see Figure 3). On the Internet, unicasting is performed using UDP (User Datagram Protocol) messages for unreliable unicasting and TCP (Transmission Control Protocol) messages for reliable unicasting.

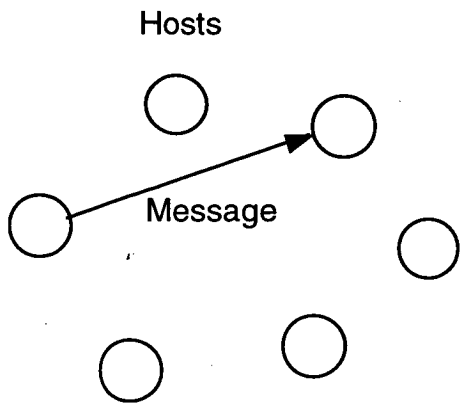


Figure 3 Unicasting

An important factor associated with message sending is network latency. *Network latency* is the time it takes for an empty message to travel across a network from the source application layer to the destination application layer (see Figure 4). Latency will always be present due to the speed of light limitations.

Latency has been ignored for a long time due to the nature of the current generation of networked applications that shunt relatively large amounts of data across the network. With the advent of more interactive and group oriented networked applications, latency is becoming recognized as the other significant limiting factor in networks besides bandwidth.

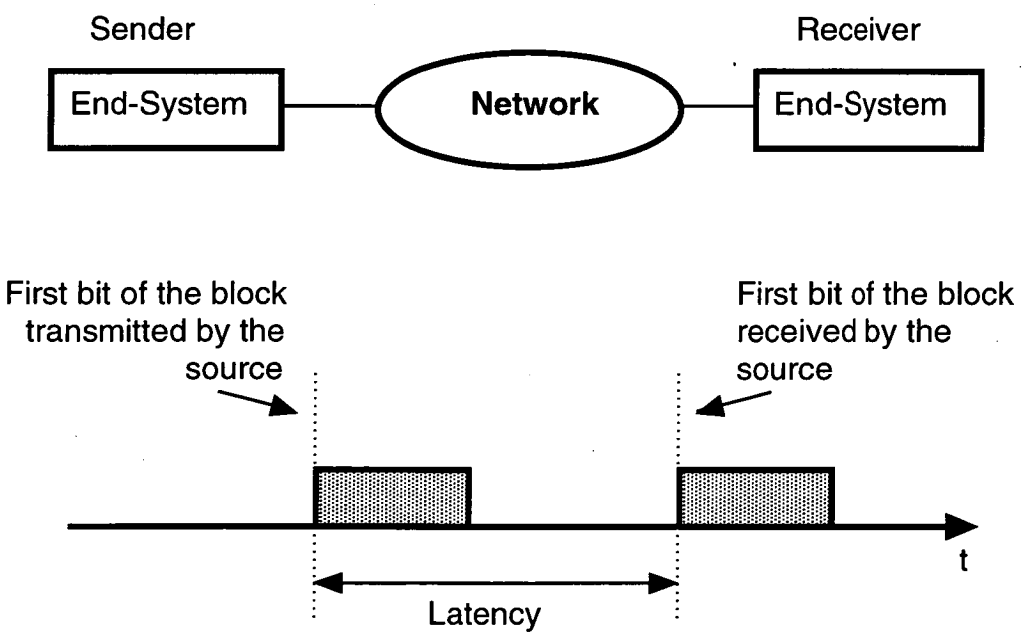


Figure 4 Network Transit Delay or Latency (Fluckiger 1995)

A more useful measure than latency for assessing the performance of multi-user virtual reality systems is end-to-end delay. This measure also takes into account the time it takes for the data to be transmitted onto the medium and received from the medium, not just the transit time (see Figure 5).

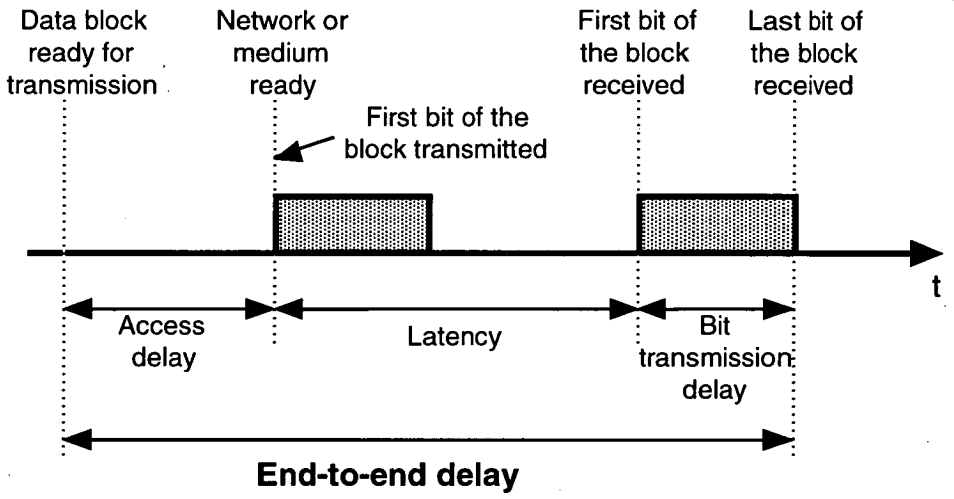


Figure 5 End-to-End Delay (Fluckiger 1995)

Detailed definitions of the components of Figure 5 are given below,

- The *access speed* is the frequency at which bits may be sent or received during transmission periods at the interface between the end-system and the network. As not all networks can transport bits at that speed in a sustained way over long periods of time, the achievable average bit rate may be lower than the access speed. In general, packet networks do not transport data at the access speed, whereas all circuit-based networks can (Fluckiger 1995).
- The time necessary at the source to wait for the medium to be available or for the network to be ready to accept the block of information is sometimes called the *access delay* (Fluckiger 1995).
- *Bit transmission delay* is the time necessary to actually transmit the sequence of bits of the block, one after the other, once the network is ready. For a given block size, this delay only depends on the access speed (Fluckiger 1995).
- The *return trip delay* or *round trip time* is the elapsed time between the emission of the first bit of a data block and its reception by the same end-system after the block has been echoed by the destination end-system. This includes the time taken by the destination end-system to receive, store and retransmit the block. Thus it cannot be entirely taken as an intrinsic characteristic of the communications subnetwork (Fluckiger 1995).

Note that reducing a message's size not only reduces the bandwidth required by a multi-user virtual reality system, but also the end-to-end delay by reducing the bit

transmission delay; as a consequence, data compression can reduce end-to-end delay. Data compression can also lead to increased end-to-end delay if buffering is used by the data compression scheme. Compression should thus be done on a per message basis. An example of where data compression increases end-to-end delay is data compression on modems.

Typically, end-to-end delays of less than 100ms are required for audio interaction and less for interactive manipulation requiring visual feedback. Other properties such as object motion might be less sensitive to latency, especially if described by behaviours evaluated locally at each peer (Hagsand 1996).

For networks with large latencies, such as Wide Area Networks (WANs), the network latency becomes a critical issue in the performance of the system. Local Area Networks (LANs) typically have network latencies less than 10ms and so can use relatively inefficient algorithms in terms of response and notification times. WANs can have network latencies of about 12.5ms to 400ms (Hagsand 1996).

Table 1 shows some typical latencies obtained from experimental tests using 'traceroute' (see a Unix manual for more information on this utility) and from Cheshire 1996 and Hagsand 1996.

Round Trip Time	*Latency*	*Route*	*Source*
1 to 3ms		Within a LAN	traceroute
10ms		ISDN to ISP	Cheshire 1996
260ms	130ms	Modem to ISP	Cheshire 1996
	26ms	Computer's Serial Port to Modem	Cheshire 1996
	50ms	Internal Modem Delays	Cheshire 1996
	15ms	Screen Redraw Delay (75Hz)	Cheshire 1996
	variable	Local Host Processing	
	28ms	Telephone Line Delay	Cheshire 1996
10ms		Tas Uni within Tasmania	traceroute
30ms		Tas Uni within Australia	traceroute
300ms		Tas Uni to the USA	traceroute
400ms		Tas Uni to Europe	traceroute
500ms		Tas Uni to Japan	traceroute
600ms		Tas Uni to Africa	traceroute
25ms		Within Sweden	Hagsand 1996
200ms		Sweden to America	Hagsand 1996
800ms		Sweden to Japan	Hagsand 1996

Table 1 Typical Latencies

From this information, it can be seen that latency must be reduced at all costs in WANs and when modems are used. For LANs, latency is not a major issue. If WANs are being used it is necessary to design the software architecture so that response, notification and commit times are optimal because that 100ms limit may already have been exceeded.

Reliable Unicasting

With reliable unicasting, the delay for a lost packet using TCP is typically (Tanenbaum 1996):

$\tau_{eru}[ab]$ = timeout + $\tau_{eu}[ab]$

where

$\tau_{eru}[ab]$ = end-to-end delay for the reliable unicast from host 'a' to host 'b'

timeout = $\tau_{rtt}[ab] + 4.\ \tau_{dev}$

$\tau_{eu}[ab]$ = end-to-end delay for an unreliable unicast from host 'a' to host 'b'

where

$\tau_{rtt}[ab]$ = round trip time between hosts 'a' and 'b'

= $2.\tau_{eu}[ab]$, assuming the end-to-end delay is symmetrical

τ_{dev} = mean deviation of $\tau_{rtt}[ab]$

≈ standard deviation of $\tau_{rtt}[ab]$

The reader is referred to Tanenbaum (1996) for the exact method for calculating 'τ_{dev}'. Substituting, we find that:

$\tau_{eru}[ab]$ = timeout + $\tau_{eu}[ab]$

= $(\tau_{rtt}[ab] + 4.\tau_{dev}) + \tau_{eu}[ab]$

= $4.\tau_{dev} + 3.\tau_{eu}[ab]$, assuming end-to-end delay is symmetrical

≥ $3.\tau_{eu}[ab]$

Thus a reliable unicast sent via TCP that needs to recover from a packet loss will incur a delay at least three times larger the normal end-to-end delay.

There is a trade-off between the number of duplicate messages and the recovery time from dropped messages. The more duplicate messages that are sent, the better the recovery time from dropped messages. The fewer duplicate messages, the worse the recovery time from dropped messages. The correct balance for the application at hand needs to be found. For multi-user virtual reality applications for instance, using TCP for reliable unicasts would not be optimal if there was spare bandwidth available (e.g. on a dedicated LAN with not many hosts). A protocol that sent more duplicate messages would be more appropriate.

Multicasting

Multicasting is the process of sending a single message to a group of hosts on a network. This is different to unicasting where a single message is sent from one host to one other host (see Figure 6). If multicasting is implemented in hardware (sometimes it is emulated by software using multiple unicasts) it can dramatically reduce the bandwidth required to send a single message to multiple hosts—a task required very often in multi-user virtual reality systems and other groupware applications. When the term 'multicasting' is used, it usually refers to hardware-supported multicasting.

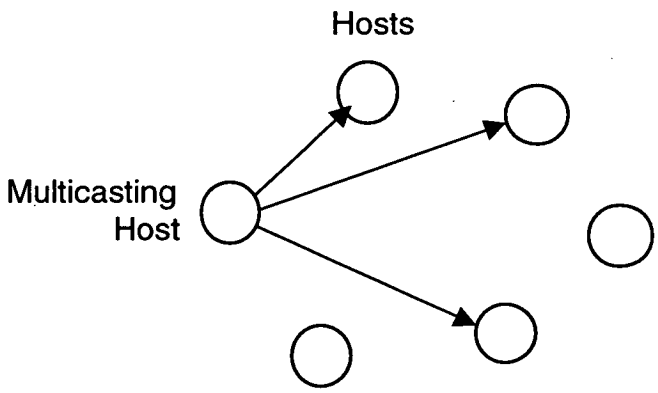


Figure 6 Multicasting

Local host processing requirements are also reduced over unicasting. The sending host only needs to send one message (unlike unicasting where a new message must be sent to each host). If the number of members of a multicast group is 'n' then n-1 messages would have to be sent for a unicast implementation of multicasting (you do not have to send one to yourself), whereas hardware multicasting requires only one.

A host will request to become a member of a particular multicast group. If the request is successful then that host will be able to receive and send messages to that particular group. It should be noted that the time required to join and leave multicast groups is finite, as is the number of available multicast groups.

Multicasting is supported over the Internet via the IP Multicast standard as described by Deering (1989). The limited number of multicasts groups available on the Internet is a restriction for multi-user virtual reality systems (Hagsand 1996 p. 36, p. 38). There can be difficulty managing which applications use which addresses in the multicasting address space since there is no register of which ones are being used at any given time.

A major downfall of multicasting is that it is not supported everywhere on the Internet yet. It is supported over the MBONE (multicast backbone), of which not everyone on the Internet is a member (although anyone can request a multicast feed). Currently the MBONE bandwidth is limited to 500Kbps and the size of a message is limited to

approximately 50 bytes (Gautier & Diot 1998). Due to this limited bandwidth, sessions on the MBONE may have to be booked (Savetz et al 1998).

A consortium of major computer industry players has joined the IP Multicast Initiative (http://www.ipmulticast.com/), including Microsoft and Intel. They are attempting to accelerate the adoption of multicasting to support the emerging generation of multicast enabled applications. These include replicated databases and groupware applications.

Reliable Multicasting

Within multi-user virtual reality systems, it is often required that messages are sent reliably. For this reason reliable multicasting and its effects on performance and scalability are discussed here.

Several attempts have been made to make generic reliable multicasting schemes, but it has been recognized that different applications have different requirements and can suffer severely from using a generic multicast protocol that accommodates for the most demanding application (Floyd et al 1995). For example some applications require total ordering of messages, whereas some require no ordering at all. Ordering severely decreases the performance of a reliable multicast protocol.

One philosophy is to build a multicast protocol that meets only the minimal requirements of eventual delivery of every message to its destination. If further ordering is required, it is possible to layer this on top of the minimal reliable multicasting scheme.

In order to have reliable multicasting some sort of acknowledgment mechanism is necessary. Reliable unicasting uses positive acknowledgments: when a receiver receives a message, the fact is acknowledged. This mechanism does not scale well when applied to multicasting because as the number of hosts increases, so does the number of acknowledgments when a message is sent.

An alternative scheme is to use negative acknowledgments. Each message that is multicast is tagged with a sequence number by the sender. Receiving hosts keep track of the sequence numbers and can detect when a message is missed when it receives its next message because messages will arrive out of order. When the missing message is detected, a negative acknowledgment (NAK) is sent to the sender for the message to be resent.

Negative acknowledgments can suffer from the problem of NAK implosions. These occur when a message is lost near the source. Because a large number of receivers will not receive the message, a large number of NAKs will be sent. The way around this problem is to use a local recovery mechanism (see Floyd et al 1995).

An ideal local recovery mechanism would ensure that when a receiver lost a message it would always be recovered from the nearest peer with the missing message. A well-designed reliable multicast protocol will have lower recovery times than a reliable

unicast protocol because recovery can occur from a closer peer that received that data correctly, not just from the original sender (Holbrook et al 1995, Floyd et al 1995):

$\tau_{erm}[ab] \leq \tau_{eru}[ab]$

where

$\tau_{eru}[ab]$ = end-to-end delay for a reliable unicast from host 'a' to host 'b'

$\tau_{erm}[ab]$ = end-to-end delay for a reliable multicast from host 'a' to host 'b'

This advantage will be seen most in a topology where hosts are more likely to be significantly closer (in terms of network delay) to neighbouring peers than the original sender (see Figure 7). Star, Bus and Complete topologies will see the least difference between reliable multicasting and reliable unicasting (see Figure 8).

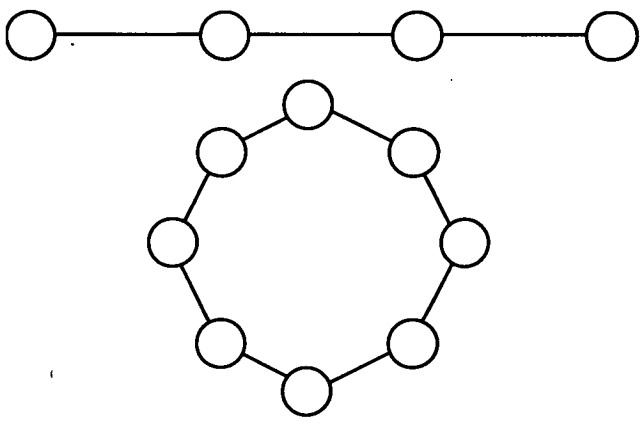


Figure 7 Topologies where Neighbouring Hosts are Likely to be Closer than the Sender

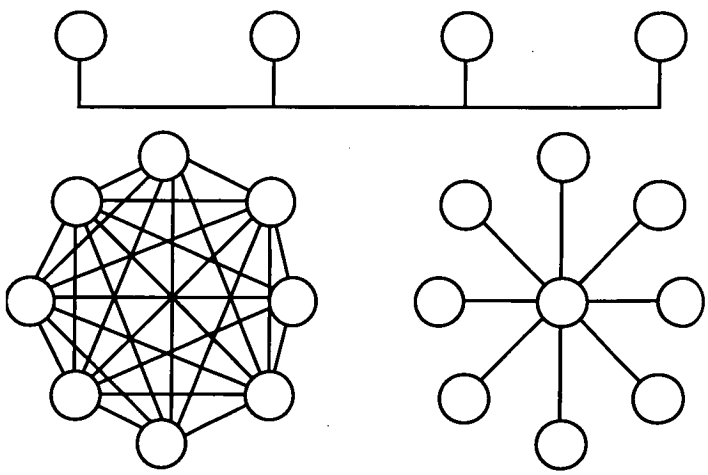


Figure 8 Topologies where Neighbouring Hosts are Not Closer than the Sender

A reliable multicasting standard is not defined for the Internet, but one can implement reliable multicasting on top of IP multicasting and unicasting facilities.

As with reliable unicasting, there is a trade-off between the number of duplicate messages and the recovery time from dropped messages. Log-Based Receiver-Reliable multicasting for instance (Holbrook et al 1995) actually sends duplicate messages but has smaller recovery times.

Broadcasting

Broadcasting is a special case of multicasting where all hosts receive messages sent. This type of network traffic is generally unfriendly because hosts do not have a choice but to process each message that is broadcast. This takes up processor time if they are not interested in the messages. Broadcasting has, however, been used for military simulations running over dedicated networks where each connected host has been part of the same simulation, thus interested in every message (see SIMNET and DIS).

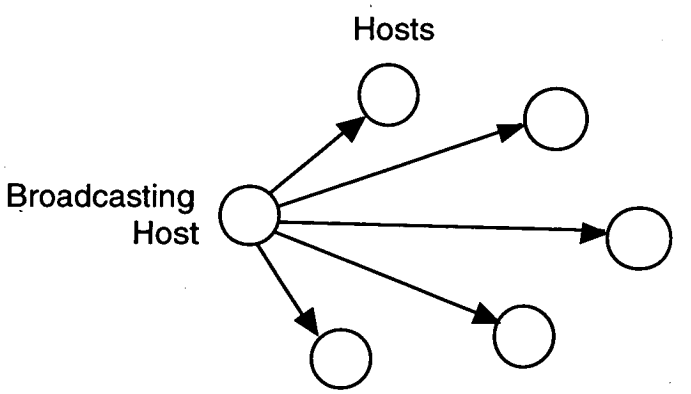


Figure 9 Broadcasting

Message Passing Summary

Multicasting is better when sending to a large number of peers because it has faster recovery from lost packets, consumes less bandwidth and less processing time. Unfortunately it lacks the hardware infrastructure on the Internet that unicasting has.

Unicasting can be used when sending messages between single peers because it will consume less processing time and less bandwidth.

Broadcasting should only be considered when a dedicated network is being used and where all hosts are interested in all the messages on the network. Generally, it ought to be avoided.

2.2 Remote Entity Approximation

Multi-user virtual reality usually requires some degree of object replication: Copies of objects are simulated on local hosts so that they are available for rendering and smooth animation. Remote Entity Approximation deals with the synchronization of these locally simulated objects and interactions from remote users.

Dead-Reckoning

In multi-user virtual worlds, there are special entities called 'avatars'. Avatars are special in that they are visual representations of users in a virtual world. They are tightly coupled to the user's input devices, and usually change very frequently in a non-deterministic manner in response to the user's input. Avatars can be any object in a virtual world, but are usually some sort of vehicle (such as a tank in a military simulation) or a humanoid object.

It is difficult to increase the number of avatars in a virtual world due to the unpredictable nature of their actions and consequent need for updates across the network. Remote Entity Approximation takes advantage of what little predictability it can find in avatars to approximate future states of these objects in the virtual world. This predictability can come from physical constraints, such as inertia, that apply to realistically modelled objects. It reduces the frequency that these objects' positions and orientations need to be sent to other participants by sending them only after the object has deviated from its predicted path by some predetermined error (Pratt 1993). This is the principle of dead-reckoning (see Figure 10).

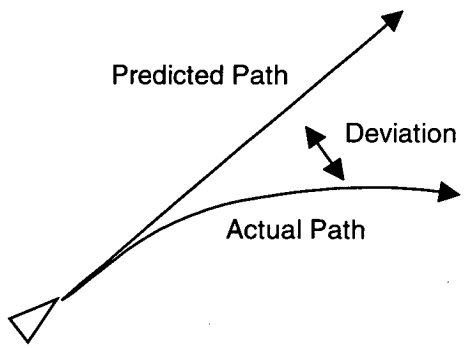


Figure 10 First Order Dead-Reckoning

Dead-reckoning uses simple parametric object behaviour prediction to reduce network bandwidth and reduce the visible effects of latency (avatar behaviour descriptions are generally simple due to their unpredictability). Due to the frequency of avatar updates and the simplicity of their behaviour descriptions, it is efficient to specify them parametrically. It then becomes a computationally simple task to predict or interpolate the avatar's future states. The position of a remote avatar with significant inertia can be

predicted at any point in time from a network message specifying the last known position and velocity of the object:

current position ≈ (old position) + (old velocity).(time since old update)

When new information arrives updating the avatar, the new position of the avatar may be significantly different to that of the predicted one. Usually smoothing techniques are used to make the transition between the old position and the new one smooth.

To update a dead-reckoned entity requires one unreliable multicast. Dead-reckoning is beneficial in handling missing network messages from unreliable delivery due to the frequency of updates: if a message is missed it will not be long before a new one is received to set the object back onto its correct course. In order to ensure a regular stream of updates so that missed messages are handled with minimum impact to the users, a time is set after which, if no updates have been sent, an update is sent anyway. A typical minimum update rate for military simulations is one packet every five seconds (Macedonia 1994). New hosts connecting to dead-reckoned simulations can also use these updates to gather information about the current virtual world state within a small period of time.

The effectiveness of dead-reckoning is determined by the predictability of the objects being dead-reckoned and the accuracy of the dead-reckoning algorithm. Some typical figures obtained from various military simulations are shown in Table 2. If dead-reckoning was not used, updates would have to be sent once per screen update to ensure smooth animation. Screen updates usually occur between 10 and 60 times per second. Taking a typical value of 30 times per second gives one update every 0.033 seconds. This means that dead-reckoning can reduce the number of updates by between 81 and 2.5 times or more typically, 10 times (based on the maximum, minimum and typical rates in Table 2).

Time between updates (secs)	Statistical Details	Source
2.7	Average from 100 updates per sec from 270 entities over 20min	Pratt 1993
0.70	Average from 142 updates per sec from ≈ 100 entities	Pratt 1993
0.30	Average from 168 updates per sec from 50 entities	Zeswitz 1993
0.083	Average from 12 updates per sec from 1 entity	Macedonia 1994
0.33	Typical Rate	Macedonia 1994
0.13	Burst Rate	Macedonia 1994

Table 2 Typical Dead-Reckoning Update Rates for Military Simulations

Using traditional dead-reckoning and broadcasting techniques for military simulation it has been estimated that in order to support 100 000 entities, being updated once per second, a sustained load of 230Mbps would be required. The peak load would reach up to 700Mbps. Using multicasting to prune the distribution of data would still require more than 100Mbps (Singhal 1996).

The above calculations also assume that the virtual world is run as the only consumer of network resources. In practice, the availability of a large dedicated network will be rare. If this technology is to be used on the Internet then it will need to perform on shared networks. These may be shared with other network services and other networked virtual worlds. The effective available bandwidth can thus be dramatically reduced. Virtual worlds will only be able to contain a fraction of the number of objects available on a dedicated network. This means that the local hosts have less information to process and can thus use some of their computational power for reducing network bandwidth (hence the Replicated Time Warp system described in this thesis).

Dead-reckoning techniques are an active area of research. The number of avatars in the system can be increased by using better dead-reckoning algorithms to reduce the network traffic, but there is a fundamental limit for non-deterministic objects because their future states cannot be precisely predicted. Variants such as Position History-Based Dead-Reckoning (Singhal 1996) have shown that improvements are still being made in this area.

The dead-reckoning techniques used above effectively reduce the bandwidth of simulated vehicles, but do not work well when applied to human models due to their erratic movements and lack of a significant inertia (Hagsand 1996). Figure 11 illustrates the possible range of paths of an avatar with significant inertia.

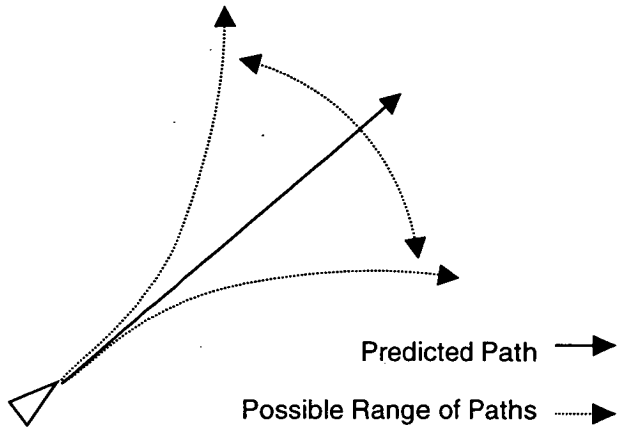


Figure 11 Limits of Prediction

If avatars were completely unpredictable, there would be no point in trying to predict what was going to happen because you would be wrong. You are better off not predicting.

Behaviours that are more complex can be used, other than simple dead-reckoning. It is unlikely, however, that very complex behaviours will be used for avatar objects because the period between updates is very small, so the avatar's behaviour only needs to be predicted into the near future.

Delay Compensation

A local host can compensate for end-to-end delay by predicting the behaviour of a remote object an extra period into the future equal to the end-to-end delay for the remote avatar. End-to-end delay can be calculated using a clock-synchronization mechanism (see section 4.3 Replicated Time Warp, Clock Synchronization).

Figure 12 illustrates the process of delay compensation. Host B sends a position update message containing information about its locally controlled avatar's current position and velocity. This is sent at time t=0 across the network to host A. The message arrives at host A at time t=2, but by this time the message is 2 time units old. Host A can predict the current position based on the last known velocity and position of the entity. This is the delay compensated position:

current position ≈ (old position) + (old velocity).(time since old update + end-to-end delay)

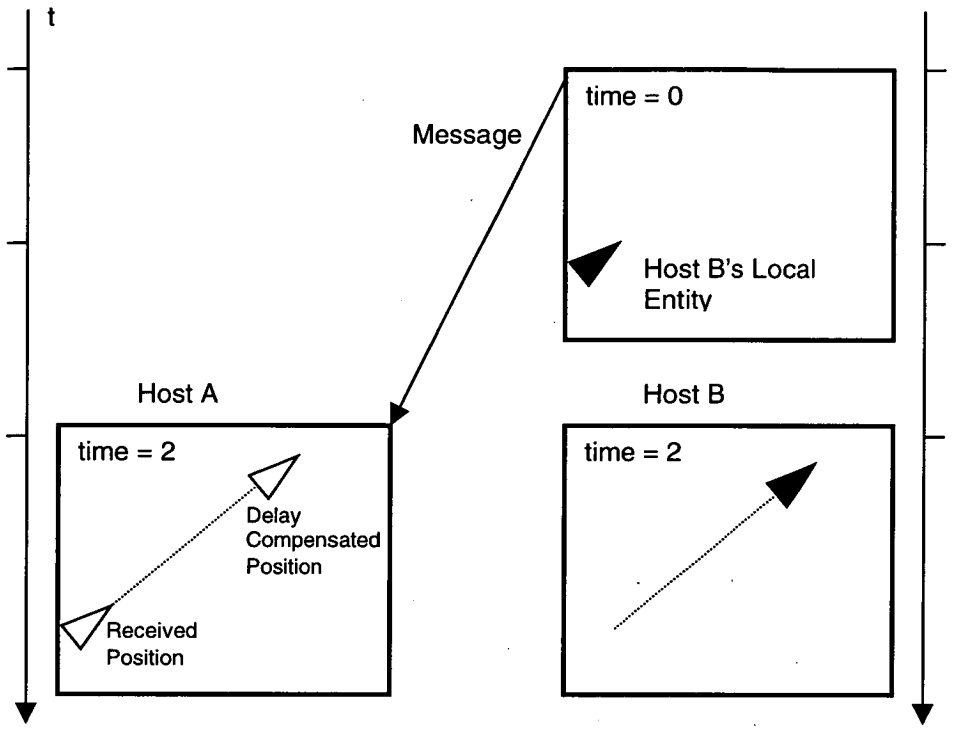


Figure 12 Delay Compensation

Locally Simulated Objects

For objects other than avatars, behaviours can be more complex because non-deterministic interactions with these objects are less frequent. Since it is not obvious which host will interact with these objects, it makes sense to simulate them in sync with the local real-time clock because then they are in sync with any local user interactions; local interactions are reflected immediately in the locally simulated object's behaviour.

Unfortunately, locally simulated objects are not in sync with remote interactions because the interactions arrive late due to network delay. This causes discontinuities in the interactions between remote avatar objects and locally simulated objects. Time Warping takes advantage of the fact that interactions are usually spatially localized around the avatar, and synchronizes locally simulated objects with nearby remote avatars (see below). If an avatar is sufficiently predictable, however, Delay Compensation is an even better way of synchronizing local behaviours and remote interactions since none of the restrictions of Time Warping are imposed (see above).

With delay compensation, the remote avatar's delay is compensated for so that interactions between remote and local entities are synchronized.

Time Warping

Time warping (not to be confused with Time Warp) is a clever technique for latency hiding for the users of multi-user virtual worlds (Sharkey et al 1996). It synchronizes the simulation time of locally simulated objects with the simulation time of remote non-deterministic objects. Normally the actions of remote non-deterministic objects in a multi-user virtual world are observed late by a local user. The actions are observed as they arrive at the local host and so have a delay equal to the end-to-end delay of the network. Because deterministic objects are often simulated locally, they generally exist synchronized with the local time. This enables interactions between the local user and deterministic objects to occur without any synchronization problems. Since the remote users' actions are observed late, interactions with the local user's copy of the deterministic objects by remote users is out of sync. This causes anomalies in the form of discontinuous jumps by the interacting deterministic objects that are visible to the local user.

Figure 13 enables visualization of the circumstances just described. This diagram consists of a flat two-dimensional plane that represents spatial coordinates in a virtual world. An object's local simulation time delay is represented on the vertical axis. Remote users exist at a simulation time that is delayed by their network end-to-end delay. Deterministic objects immediately surrounding them are simulated with zero delay, thus interactions between the remote users and locally simulated deterministic objects cause discontinuous jumps due to the discontinuous simulation times.

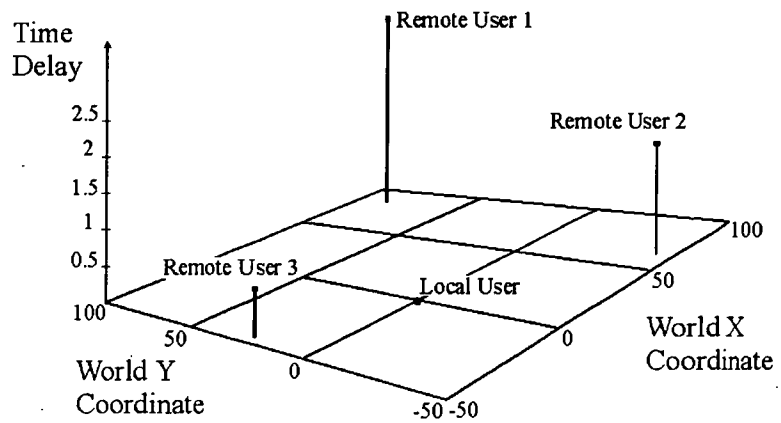


Figure 13 Remote Users Exist in a Delayed Time with Respect to the Locally Simulated Objects Surrounding Them (Sharkey et al 1996)

The time warping technique warps time within the local user's virtual world so that the local user is synchronized with the local real-time clock (as per usual), but there is a continuous graduation of time from the local-time (the 'present') to the remote user's

late time frame (the 'past') throughout the virtual world space (Figure 14). This means that deterministic objects travelling from the local user towards the remote user within the local user's virtual world actually experience a slowing of time so that when they reach the remote user's area in the virtual world, they are synchronized with the remote user's time. This means that interactions by the remote user on deterministic objects, as observed by the local user, are in sync. This means that no discontinuous jumps of deterministic objects are observable. Instead, the harsh anomalies that were present before are smoothed over time.

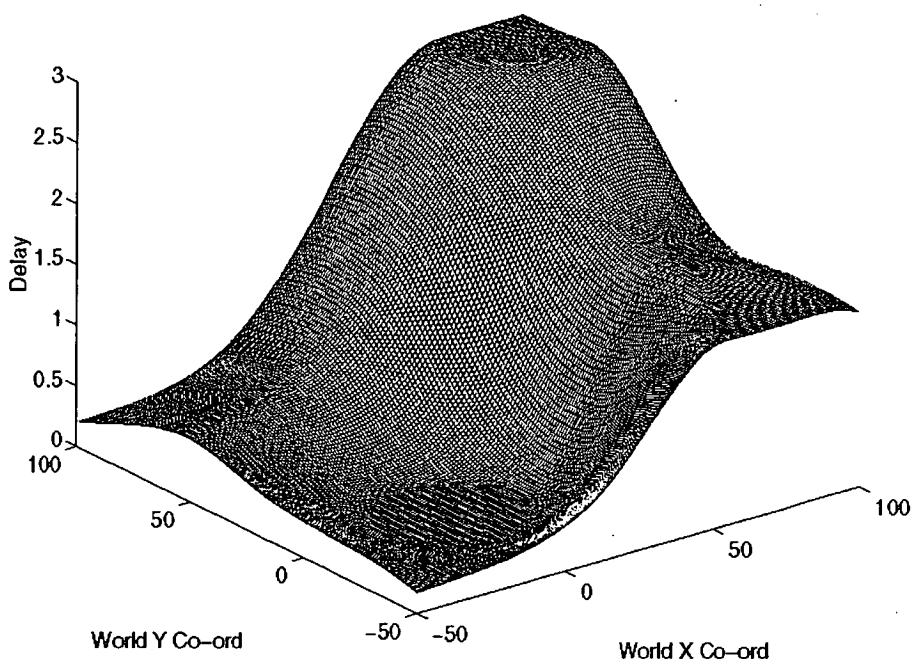


Figure 14 Remote Users Exist at the Same Time as the Locally Simulated Objects Directly Surrounding Them (Sharkey et al 1996)

An assumption of time warping is that interactions by users will be confined to a localized area—the area of space synchronized with those non-deterministic interactions. Generally, this assumption is reasonable.

An example of the effects of not having time warping versus having time warping can be considered. Consider first a simple example of a game of virtual tennis: With continuous time warping, as the local user hits the ball towards the remote user's side of the court, the ball starts transitioning to the past i.e. time slows down for the ball. The local user observes the ball approaching the remote user at a reduced speed. When the remote user hits the ball, however, the timing of the remote user's racquet swing will be synchronized with the motion of the ball and the remote user will appear to hit the ball correctly. The ball now transitions from the 'past' to the 'present' as it approaches the local user. The local user observes this as a slight speedup of the ball as it approaches. By the time it reaches the local user, the ball is now synchronized with the local user's time and can be hit correctly.

If continuous time warping was not used, the ball would always be synchronized with the local user's time all the time. As it approached the remote user, the local user would observe the ball moving past the racquet (because it doesn't know yet that the remote user has swung at it). When the data arrives that tells us that the remote user has swung their racquet, rollback of the ball occurs, the racquet swing event is inserted and the ball's position is recalculated (the rollforward). The end result is that the ball jumps from behind the remote user's racquet to the correct position in front of the racquet. The ball has performed an ugly discontinuous jump, rather than a nicer smooth transition.

2.3 Data Distribution

One of the main issues to decide upon when designing multi-user virtual reality systems is the data distribution mechanism. There are two main data distribution mechanisms: client-server and peer-to-peer (or replicated). The choice of data distribution mechanism affects the performance and scalability of the multi-user virtual reality system. The following discussion assumes the availability of hardware multicasting for message passing.

Client-Server

Client-server multi-user virtual reality systems typically consist of a server that controls the initial distribution and updating of the virtual world for the connected clients (see Figure 15). Clients can request a connection to the server. Once granted, the server then transfers the necessary data for the client to be synchronized with the other clients. The client is added to the list of clients that need to be updated by the server.

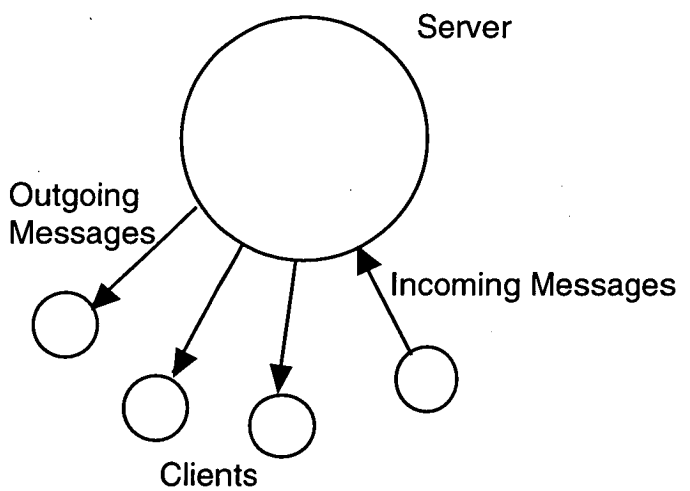


Figure 15 Client-Server Architecture

Topology Independence

The client-server data distribution mechanism is naturally suited to a star network topology. This topology is most commonly seen on the Internet where a server is set up to allow users dial-up access to the Internet over a modem.

Client-server architectures can increase delay times and bandwidth requirements because all changes to data need to be made via the server (Waters et al 1996). The delay becomes that of the time taken for the message from the client to reach the server, the time taken by the server in receiving the message from the network and processing it if necessary, and the time taken for the server to send the message to the other clients. In the case of a star network topology, the client-server notification time can equal the peer-to-peer notification time because all traffic has to go through the server at the centre of the star anyway. The server can either unicast the message to the other clients or, more efficiently, multicast it.

The client-server architecture can work well when network delays are very small, such as on a LAN. It can also result in a simpler overall implementation. Because all operations performed on objects in the virtual world are transferred via the server, the server is the only computer that needs to deal with concurrency control issues. The client-server architecture simplifies concurrency control because the data is maintained at a central location. Consistency requirements are easily met since the data is modified centrally and then distributed to clients.

Security

Having data centrally located, it is also easier to keep data secure and to prevent hacking of the data (Intel 1997). The server can perform operations on the data and have the results distributed to the clients, rather than having calculations on the data performed by peers (where the chance of hacking of data is possible) and then having the results distributed. Hacking of data can only occur through operations sent across the network, but these are filtered by safe methods that only change the data in a legal manner (Figure 16).

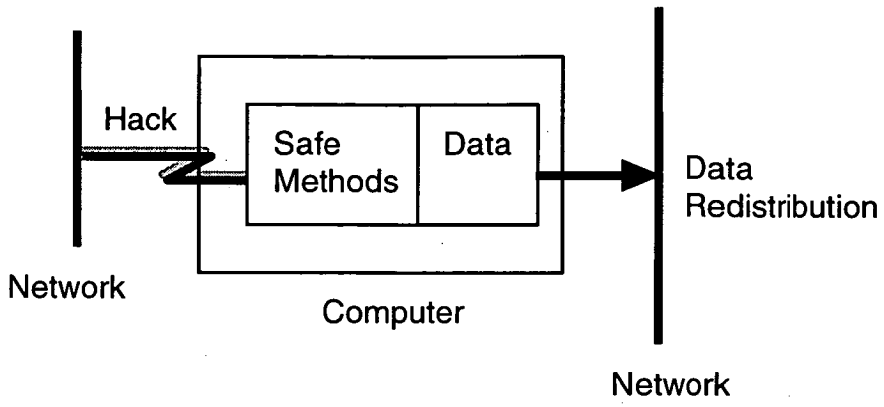


Figure 16 Client-Server Security

Scalability

The client-server architecture does not scale well in terms of the number of users it can support because as the number of users rises, the server must process more messages per second. For each client, the server must receive the client's update, process it and distribute the results.

The client-server architecture also requires a larger bandwidth than peer-to-peer (except in a star topology) because a message is sent to the server and then from the server to the clients, rather than directly from the local peers to remote peers in the peer-to-peer scheme.

Fault Tolerance

The server in a client-server architecture is also a critical point of failure. If the server fails then none of the clients can operate in a multi-user manner any longer. This is because the server is relied upon both to distribute client messages to other clients and for concurrency control.

Partitioning Ability

Partitioning (also known as 'replication on demand' or filtering) is the breaking up of a virtual world into different sections based on a particular host's interests in remote events. These partitions are ideally created such that they are independent of each other. This reduces the amount of information that a particular host needs to know and thus also reduces the required network bandwidth and local host processing. The host attempts to operate purely on a 'need-to-know' basis. In an optimal application hosts only receive exactly the data of interest to them (Macedonia 1996).

Typical partitioning seen in virtual worlds includes 'rooms' where one needs to pass though some kind of doorway to observe or interact with the next room and its contents. Virtual worlds can also be partitioned into large areas based on the maximum viewing distance of a host. If you cannot observe a distant part of the virtual

world, you do not need to know about it (assuming also that the host cannot interact with anything it cannot see).

Another form of partitioning involves sending out various levels of detail of information. If a host has good bandwidth it can receive the maximum amount of information. If the host has limited bandwidth it can receive a courser level of information that requires less bandwidth.

In the client-server situation, because the server has complete knowledge of the virtual world, messages can be precisely filtered to individual clients. For example, if the server is powerful enough, it can calculate which objects a client can see and send information only regarding those objects. This is particularly useful if a client has limited bandwidth because the server can apply flow control strategies to ensure the network does not become congested. Congestion causes packets to be lost, thereby introducing a much larger delay when trying to recover from the lost packets by retransmission mechanisms.

Peer-to-Peer

For peer-to-peer (or replicated) data distribution the virtual world is replicated (either partially with the use of partitioning, or fully) at the respective hosts. For virtual reality applications, replication is always used to some extent because the state of the virtual world is required at each local host for rendering. It is difficult to hold the visible portion of the virtual world at a remote location when it needs to be accessed around 30 times per second for rendering.

The replication distribution mechanism makes it harder to maintain consistency between data in virtual worlds, but avoids the server bottleneck that a client-server architecture has. It can also produce optimal application notification times.

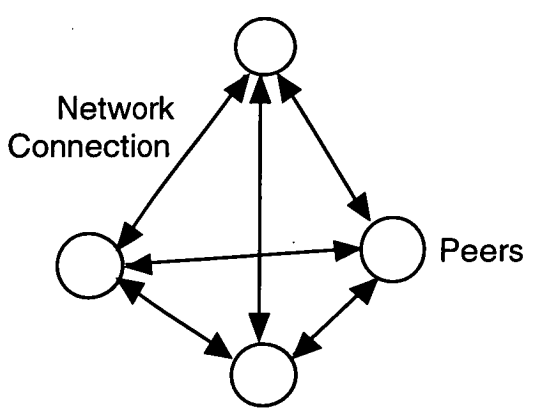


Figure 17 Peer-to-Peer Architecture

Topology Independence

The peer-to-peer architecture operates well in any network topology. This is because messages are sent directly from peer to peer (see Figure 17) and thus depend only upon the network's routers to reach their destination. They should reach their destination using the optimal route if the network's routers do their job.

Security

Having data altered and distributed by peers increases the chances of hacking occurring because the peer can directly alter data held within the computer or it can send false information about the data when it redistributes it (Figure 18).

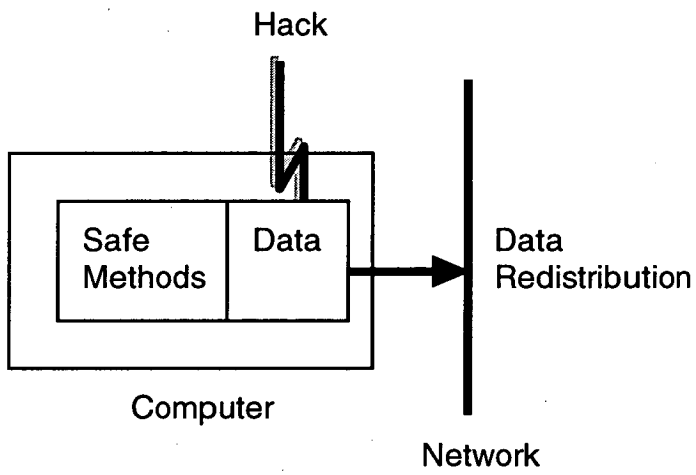


Figure 18 Peer-to-Peer Security

Scalability

The peer-to-peer architecture scales better than the client server architecture when multicasting is available because messages are sent directly to peers (reducing required bandwidth). The processing load is distributed more evenly because each peer calculates its own next state and distributes it itself.

Fault Tolerance

If a peer fails, other peers can continue operating because they do not depend on the operation of each peer. The failed peer can either re-enter the virtual world (i.e. resynchronize its entire virtual world database) or it may be able to recover smaller portions of lost information from nearby peers.

The peers that did not fail will not receive any more updates from the failed peer, so the objects that were the responsibility of that peer to update will no longer be updated. This is not a fatal circumstance—these objects can be removed from the

virtual world after some timeout if required or another peer may be able to take over responsibility for updating these objects.

Partitioning Ability

Partitioning in a peer-to-peer situation is more complex than the client-server scenario because peers suddenly do not have a complete picture of the virtual world when partitioning occurs. This is because as soon as a peer stops sending information to another peer, that peer loses track of the peer and cannot work out whether to send updates to that peer or not. Instead, the virtual world needs to be partitioned in agreed upon partitions. Each partition would typically be assigned a different multicast address so that anyone in that partition could join that multicast group and receive information relevant to that partition. Because each partition only receives information about partitions it is interested in, as opposed to information concerning the entire virtual world, bandwidth can be reduced.

Hybrid Systems

A combination of both client-server and peer-to-peer data distribution mechanisms can be used where appropriate (see Figure 2-19). Hybrid systems can use the benefits of both models, depending on the network that is already in place. For example, on the Internet it is often the case that modem pools exist. These conform to a star network topology. This is an ideal place for a client-server solution. The servers that the modems connect to are good candidates to be linked through a peer-to-peer mechanism.

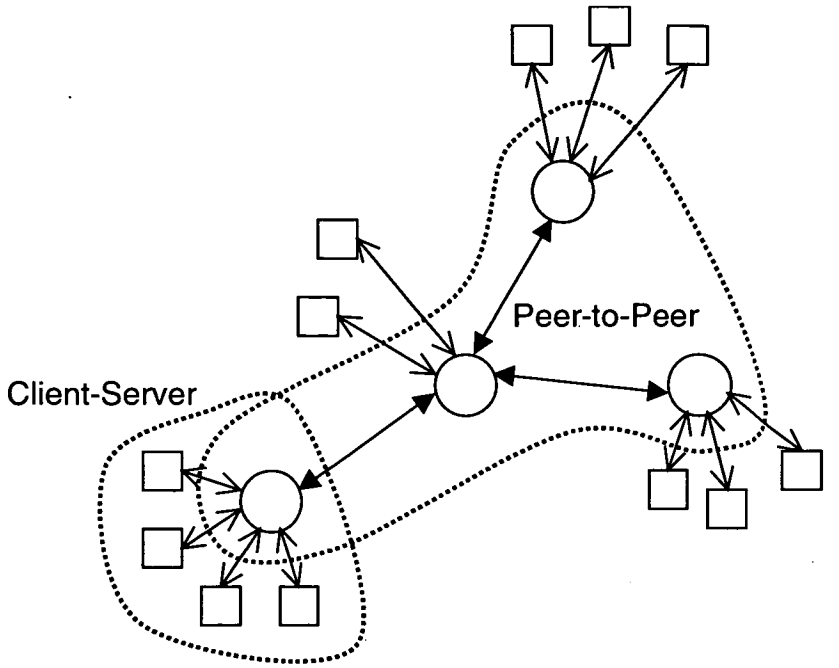


Figure 2-19 Hybrid Architecture

The following benefits are found that are specific to client-server modem pools:

- Multicasting does not work over popular modem protocols (e.g. PPP). Unicasting is just as efficient in the Client-Server model.

- Maximum security is maintained. Hackers will find it difficult to compromise the security offered by the client-server solution.

- Maximum partitioning can be applied. This ensures that minimum network bandwidth is used, which is particularly useful for limited bandwidth devices such as modems.

The communications between the servers that host the modem pools are good candidates for peer-to-peer communication. The reasons are:

- Security is not as much of an issue because the servers are usually maintained in secure access areas at the Internet Service Provider. A hacker would need to access this machine in order to compromise the system.

- Because the server machines will be less likely to be moved or changed, it is easier to configure them to ensure access to IP multicasting for maximum efficiency in communication.

- Fault tolerance is provided between servers. If one server fails, the others can continue. If a server fails, all attached clients will fail.

Data Distribution Summary

Client-server systems, due to the centrally located data, have better capabilities for filtering, flow control, partitioning and security. They can also be easier to implement.

Peer-to-peer architectures provide optimal performance in all network topologies when multicasting is available, (whereas client-server provides optimal performance only in a star topology). Peer-to-peer systems also have better scalability. Peer-to-peer systems are thus good to use if the network topology is not known at the design stage or if the system is designed to work under any network topology.

	Client-Server	*Peer-to-Peer*
Fault Tolerance	Bad	Good
Topology Independence	Bad	Good
Partitioning Ability	Excellent	Good
Scalability	Bad	Good
Security	Good	Bad

Table 3 Peer-to-Peer vs. Client-Server Data Distribution

2.4 Concurrency Control

Concurrency control looks at the methods used to handle situations where multiple users attempt to modify the same object at the same time. Concurrency control enables different virtual worlds to maintain consistency between each other, and consistency enables collaboration. Very little work has been published on concurrency control for multi-user virtual reality systems.

Important performance factors associated with concurrency control are commit time (τ_c), notification time (τ_n) and bandwidth usage. Commit time and notification time are defined as follows,

- *Commit Time* — the time it takes a local user's action to be reflected locally, and to be guaranteed never to be altered.
- *Notification Time* — the time required for a user's action to be propagated to every user's interface (Ellis et al 1991)

Having a small commit time from a concurrency control scheme is important because alteration of the user's actions is disconcerting to the user. A small notification time is important to enable fast feedback between remote users to facilitate tight interaction. A low bandwidth concurrency control mechanism is important so that enough

information can be sent across the network to keep all the objects in remote virtual worlds consistent.

Many algorithms exist for concurrency control in databases, but most results do not apply to real-time groupware, such as multi-user virtual reality systems: databases are designed to give the illusion of being the only user on the system, whereas groupware systems are designed to make users aware of each other. The most important property of a groupware system, interface response time, is not as important in a distributed database design (Karsenty & Beaudouin-Lafon 1993).

Several concurrency control mechanisms are appropriate for multi-user virtual reality systems (Ellis et al 1991, Coulouris et al 1994 & Broll 1995). Some of the most appropriate mechanisms include,

- Client-Server Locking
- Peer-to-Peer Distributed Locking
- Peer-to-Peer Master Entities
- Reversible Execution

Each of these are discussed, analysed and compared. Hardware multicast capabilities are assumed to be available.

Client-Server Locking

The simplest concurrency control mechanism for the client-server architecture is a *locking* scheme. Conceptually this can be implemented using tokens. A client makes a request to the server to change the state of an object in the virtual world (Figure 20). The server then passes the client that object's token giving the client exclusive 'write' access to that object (Figure 21). If the token is not present at the server at the time of the client's request, the client must wait until the token becomes available before it can manipulate that object (Figure 21, Figure 22 & Figure 23).

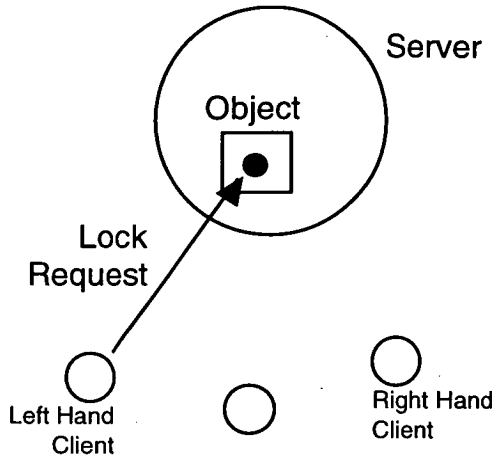


Figure 20 Lock Request

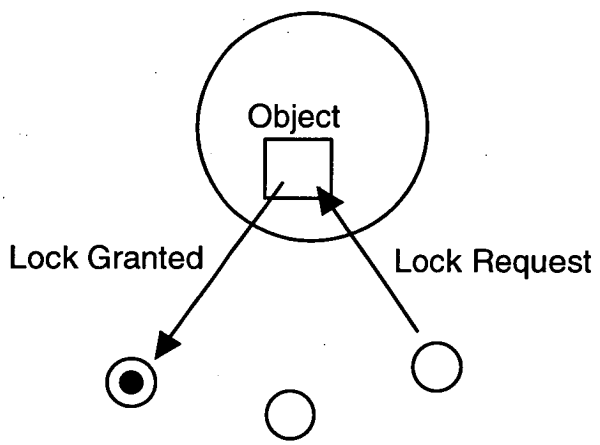


Figure 21 Lock Grant and Second Lock Request

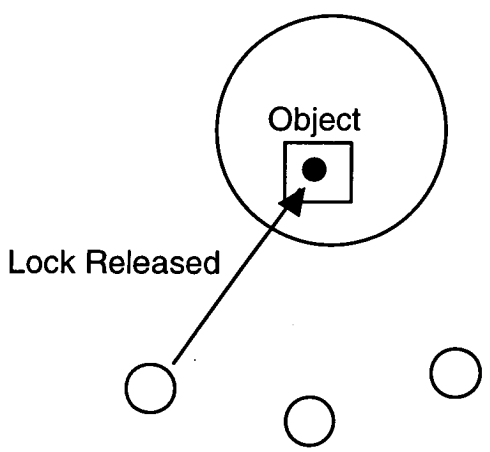


Figure 22 Lock Release

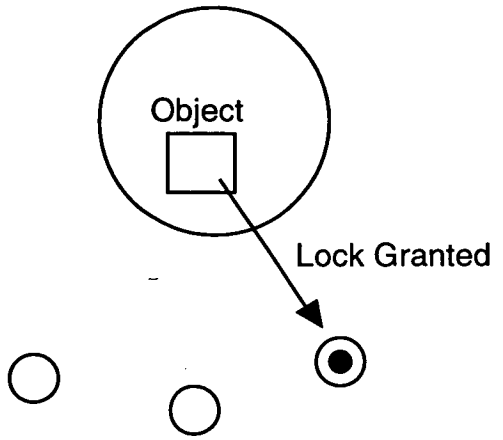


Figure 23 Second Lock Granted

Obtaining a Lock

Observing the left-hand client in the above figures it can be seen that three messages are required for each successful lock: the lock request, the lock grant and the lock release. Therefore:

n_l	$= 3.r_u$
where	
n_l	= number of messages to obtain a lock
r_u	= one reliable unicast message

The left hand client's commit time is dictated by the time it takes for the lock request to reach the server, be processed plus the time it takes for the 'lock granted' message to be sent back. In the right hand client's case the commit time is larger because the server has to wait for the lock to be released from the left hand client (tsl is larger):

τ_{cl}	$= \tau_{eru}[as] + \tau_{sl} + \tau_{eru}[sa]$
	$\approx 2.\tau_{eru}[as] + \tau_{sl}$, since $\tau_{eur}[ab] \approx \tau_{eur}[ba]$
where	
τ_{cl}	= commit time when a lock needs to be obtained
τ_{sl}	= time for the server to process the lock request
$\tau_{eru}[as]$	= client-server end-to-end delay for a reliable unicast

Once a lock has been obtained for an object, that object can be updated as many times as required without needing to obtain further permission. For an update to an object with a lock, one message is unicast to the server and the server multicasts the results to the rest of the clients:

n_u	$= 1.r_u + 1.r_m$
where	
n_u	= number of messages for an update
r_u	= one reliable unicast
r_m	= one reliable multicast

If required (i.e. if the network layer does not guarantee in order delivery of messages) the server can implement ordering by timestamping update messages with a sequence number sent to clients to ensure that they can be processed in the correct order at the clients.

Commit time is also optimal once the lock has been granted because with locking, updates are always guaranteed to be committed once the lock has been obtained:

τ_c	$= \tau_{cs}$
where	
τ_c	= commit time
τ_{cs}	= commit time for a single-user application

The notification time once a lock has been granted is the time it takes for a client's message to be passed to the other clients via the server.

$\tau_n = \tau_{eru}[as] + \tau_{sr} + \tau_{erm}[s]$

where

τ_n = notification time

$\tau_{eru}[as]$ = the client-server end-to-end delay for a reliable unicast

τ_{sr} = time for the server to relay the message

$\tau_{erm}[s]$ = the client-server end-to-end delay for a reliable multicast from the server to all clients

The time for an update to reach remote clients when a lock needs to be obtained as well is given by:

$$\tau_{nl} = \tau_n + \tau_{sl}$$

$$= \tau_{eru}[as] + \tau_{sl} + \tau_{sr} + \tau_{erm}[s]$$

where

τ_{nl} = notification time when a lock needs to be obtained

This is assuming that the update is sent in the same packet as the lock request. This is not general practice, but makes sense in multi-user virtual reality systems because the update packet is generally small. If the update were only sent after the lock has been accepted, the notification time would be that of obtaining the lock (τ_{cl}) plus the time to send the update (τ_n):

$$\tau_{nl} = \tau_{cl} + \tau_n$$

$$= (2.\tau_{eru}[as] + \tau_{sl}) + (\tau_{eru}[as] + \tau_{sr} + \tau_{erm}[s])$$

$$= 3.\tau_{eru}[as] + \tau_{sl} + \tau_{sr} + \tau_{erm}[s]$$

Discussion

The main disadvantage of this concurrency control mechanism is that the server becomes a bottleneck as the number of lock requests increases. This happens because all requests from clients are passed through and processed by the server. The network connection to the server has to have a high enough bandwidth to handle the quantity of traffic from incoming and outgoing messages. The server must also have enough processing power to process all the client requests. In addition, as with other client-server systems, the server is a critical point of failure.

The server can be designed such that the main programming logic is located on the server only. Clients send requests for operations to be performed by the server and the server processes them and reports the results back. This aids application development because program updates only need to be deployed at the server. Having program logic in one location also makes security tighter.

Peer-to-Peer Distributed Locking

The peer-to-peer distributed locking scheme is similar to the client-server locking scheme except that in this case the lock request is multicast from the requesting peer to each other peer:

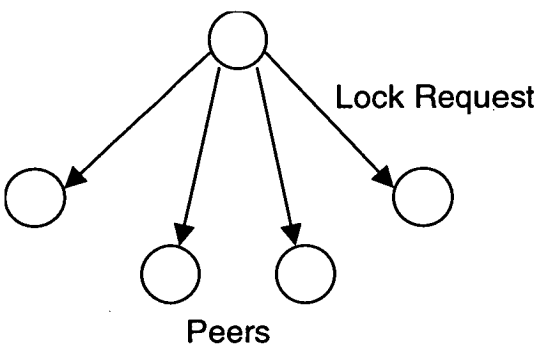


Figure 24 Lock Request

n_{lrq}	$= 1.r_m$
where	
n_{lrq}	= number of messages for a lock request
r_m	= one reliable multicast

Peer-to-peer distributed locking requires that each host acknowledge the lock request before the lock is granted:

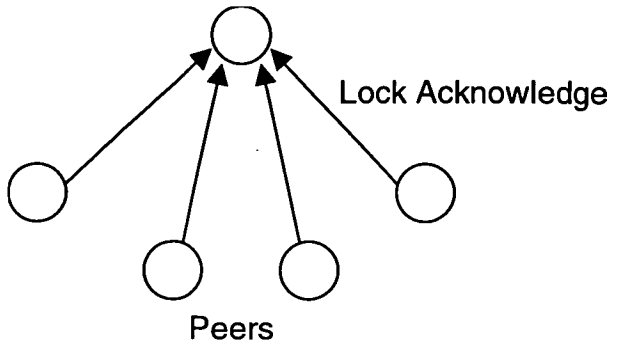


Figure 25 Lock Acknowledge

n_{la}	$= (n_p-1).r_u$
where	
n_{la}	= number of messages for a lock acknowledge
n_p	= number of peers

When the lock is released, each peer must be notified:

n_{lr}	$= 1.r_m$

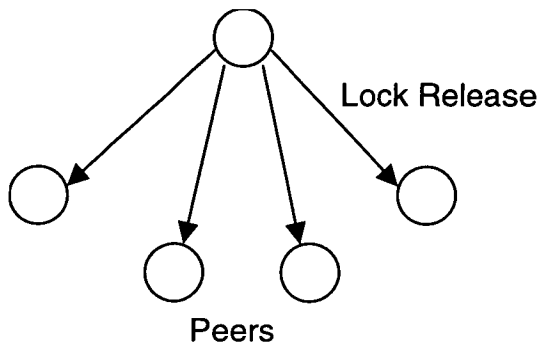


Figure 26 Lock Release

The total number of messages that must be passed to obtain a successful lock is thus:

$n_l = n_{lrq} + n_{la} + n_{lr}$

$= 1.r_m + (n_p-1).r_u + 1.r_m$

$= 2.r_m + (n_p-1).r_u$

where

n_l = total number of messages for a lock

The commit time (the time to get a lock),

$\tau_{cl} = \max(\tau_{erm}[an] + \tau_{sln} + \tau_{eru}[na])$

where

τ_{cl} = commit time when a lock needs to be obtained

τ_{sln} = time for the peer 'n' to process the lock request

$\tau_{eru}[na]$ = peer-to-peer reliable unicast end-to-end delay for from peer 'n' to peer 'a'

$\tau_{erm}[an]$ = end-to-end delay for the reliable multicast message from peer 'a' to peer 'n'

With locking, updates are always guaranteed to be committed once the lock has been obtained. Commit time is optimal once the lock has been granted:

$\tau_c = \tau_{cs}$

where

τ_{cs} = commit time for single-user application

The notification time once a lock has been obtained is the time it takes for a client's message to be passed to the other clients:

$\tau_n = \tau_{erm}[a]$

where

$\tau_{erm}[a]$ = the peer-to-peer end-to-end reliable multicast delay for reliable multicast from peer 'a' to all peers

Each update requires one reliable multicast message:

$$n_u = 1.r_m$$

The notification time when a lock has to be granted as well is given by:

$$\tau_{nl} = \tau_{cl} + \tau_n$$

$$= \max(\tau_{erm}[an] + \tau_{sln} + \tau_{eru}[na]) + \tau_{erm}[a]$$

Discussion

Peer-to-peer locking does not scale very well due to the number of acknowledgments that are needed: one from each remote peer.

Peer-to-Peer Master Entities

In this scenario, a special instance of each object exists called the master entity. The master entity is responsible for updating data about itself at other hosts so that all interested hosts can observe it. The holder of the master entity has the exclusive rights to change that entity (although other peers can request modification). Other processes can only modify the object by obtaining the master entity through migration (transfer of the entity across the network), or by sending requests to the master entity. The master entity can then update itself based on these incoming messages. The changes to the master entity from remote hosts are serialized (total ordering) by the master entity.

The process of obtaining the master entity is a similar process to obtaining a lock in the client-server locking protocol. First, a peer that wants control of the master entity requests the entity from whoever has it:

$$n_{lrq} = 1.r_u$$

The peer with the master entity can then transfer it, or a token representing it if the peer already has a replicated version of the master entity:

$$n_{la} = 1.r_u$$

The total number of messages for a successful transfer is thus:

n_t	$= n_{trq} + n_{ta}$
	$= 2.r_u$
where	
n_t	= total number of messages for a successful transfer

Updates to the master entity are then transmitted directly to peers from the holder of the master entity:

n_u	$= 1.r_m$

The holder of the master entity enjoys optimal commit time when performing operations on that object:

τ_c	$= \tau_{cs}$
where	
τ_c	= commit time
τ_{cs}	= commit time for a single-user application

The commit time for the first update (when master entity is not already at the local host):

τ_{cl}	$= \tau_{eru}[ap] + \tau_{st} + \tau_{eru}[pa]$
	$\approx 2.\tau_{eru}[ap] + \tau_{st}$, since $\tau_{eru}[ap] \approx \tau_{eru}[pa]$
where	
τ_{cl}	= time to obtain the master entity
τ_{st}	= time for the peer to process the master entity transfer request
$\tau_{eru}[ap]$	= reliable unicast delay from peer 'a' to master entity holder 'p'

The notification time is the time it takes for a peer's message to be passed to the other peers via the network:

$\tau_n = \tau_{erm}[a]$

where

$\tau_{erm}[a]$ = end-to-end reliable multicast delay from peer 'a' to all other peers

If the master entity has to be obtained first:

$\tau_{nl} = \tau_{cl} + \tau_n$

$= (2.\tau_{eru}[ap] + \tau_{st}) + \tau_{erm}[a]$

The commit time to send an update when you are *not* the holder of the master entity (and you do not want to actually obtain possession of the master entity) is:

$\tau_{cu} = \tau_{eru}[ap] + \tau_s + \tau_{erm}[p]$

Discussion

The advantage of the master entity mechanism over client-server mechanisms is that the holder of the master entity enjoys the fast response times that a server can have. It also helps distribute computational load because the holder of the master entity becomes the one responsible for evolving it and distributing the resulting changes.

Initial commit times are relatively low, and bandwidths are smaller than the other locking schemes because locks do not have to be released. Remote hosts have the option of modifying master entity objects without obtaining the master entity by sending the update to the master entity. This is a good option for one-off changes because the holder of the master entity still enjoys the fast commit times.

It is possible that a peer could fail and the master entity could be destroyed. This can be avoided by replicating the master entity at each peer and passing a token that represents the holder of the master entity. If a peer fails, the responsibility of managing the master entity can be passed to another peer.

Reversible Execution

Reversible execution is the most unusual of the concurrency control mechanisms looked at in this section. It is the only optimistic concurrency control mechanism. It also has a lot in common with the Replicated Time Warp mechanism presented in this thesis. Three factors characterize reversible execution,

- Operations are executed immediately (otherwise known as 'optimistically').
- Operations are globally ordered.

- When two or more operations have been executed concurrently, one or more of these operations may have to be undone and re-executed in the correct order.

The 'undoing' of conflicting operations is also known as *rollback*, and the 're-execution' of operations is known as the *rollforward* process. The entire process is called 'conflict resolution'.

Some researchers have stated that they think that reversible execution concurrency control is not suited to virtual reality systems because of the real-time nature of VR systems and the extra time required to perform rollbacks (Broll 1995, Macedonia & Zyda 1995). Other researchers (Sarin & Greif 1985, Karsenty & Beaudouin-Lafon 1993) think that reversible execution is a good idea for real-time cooperative work because of its fast response times. Others are undecided (Wang et al 1995). Ultimately the use of reversible execution should not be accepted or rejected immediately, but be decided upon on a case-by-case basis.

A controversial side effect of reversible execution, caused by the fact that operations are executed immediately upon receipt, is that it does not always provide the user with a 'what you see is what it is' view of the simulation. A participant may see inconsistent states that no other participant sees and that could not have occurred if command execution were delayed until the correct order were known. The application designers must consider whether temporarily inconsistent states are an acceptable price to pay for the improved response time (Sarin & Greif 1985).

Karsenty & Beaudouin-Lafon (1993) describe an algorithm that is essentially reversible execution for concurrency control in groupware application. Within their system, each object is considered independent of each other object. This means that any incoming events can be executed in any order and the results will remain the same. In most multi-user virtual reality systems this will not be the case: objects will have dependencies between each other. It is necessary to eliminate as many dependencies as possible to increase efficiency, otherwise all objects dependent on the incoming event will have to be rolled back.

Shared Whiteboard applications keep a list of drawing events and execute them in order to produce the final image. When a late event arrives, the image can be redrawn with the late event executed in the correct order. This mechanism is not appropriate for larger multi-user virtual worlds because of the real-time, dynamic nature of virtual worlds. Virtual worlds have animated objects that are updated many times a second. It is not efficient to rollback every object in the simulation. A more efficient algorithm that keeps track of dependencies between objects and reduces the number of rollbacks should be used. This is what the Replicated Time Warp mechanism achieves.

Interactions caused by the local user are reflected locally immediately. The theory behind this optimistic local host processing is that most of the time the interactions caused by the local user will not conflict with interactions by remote users. Therefore, most of the time, they can be executed immediately without having to consult the other remote hosts. If conflicts do occur with remote user interactions there needs to be a mechanism in place to resolve the conflicts in a manner that attempts to be

unobtrusive for the local user. This optimism thus enables optimal response times to be obtained:

$\tau_r = \tau_{rs}$

where

τ_r = response time

τ_{rs} = response time for single user application

When a conflict arises because of a late message (due to network delay) from a remote host, the conflict must be resolved in order to keep the remote virtual worlds consistent. An example of this is if two users (at remote locations with respect to each other) attempt to move an object in opposite directions at the same time. Because of the end-to-end delay, the move operations will be performed immediately on each local host, but won't be known about at the remote hosts. For a brief moment the two virtual worlds will be inconsistent. When the information about the users' actions reaches their respective remote hosts, it is discovered that an impossible situation has occurred and the inconsistency needs to be resolved.

There are several possibilities for resolution mechanisms and the appropriate one will be chosen on a case-by-case basis, depending on the application at hand, so as to produce the least disturbing results for the end users. Possible resolution mechanisms include:

- *Host Priority* — Operations from one host (e.g. server, master entity holder or an arbitrary peer) have precedence. If conflicting operations from other hosts are generated, they are ignored.
- *Time Priority* — If events are timestamped using a unique global clock (e.g. the synchronized local clock time and the machine's IP address) then it is always possible to see which operation was performed 'first'. Whichever was performed first will be kept, and the conflicting operation that was performed second will be ignored.
- *Duplication* — The object is duplicated at each host so that all concurrent operations can be applied: one to each object copy (e.g. Singh 1995).
- *Relative Operations* — The object is not duplicated, but operations are relative, so can always be applied.

In the example with the users moving the object in opposite directions the following outcomes would occur using the different resolution mechanisms:

- *Host Priority* —The host with the highest priority will see the object being moved as they wished, but the other will see his moved object jump to where the other user moved it.

- *Time Priority* —In this case the user that performed their move 'first' will see the object being moved as they wished, but the other will see the moved object jump to where the other user moved it.
- *Duplication* — The object would be duplicated so that one copy would go in the one direction and the other in the other direction. The user now sees two copies of the object.
- *Relative Operations* — The move operations are relative, so if both users move the object in opposite directions equal amounts, the object will end up at its original position.

Another example of resolution can be seen in dead-reckoning (see Remote Entity Approximation). When a dead-reckoned entity is found to have followed a path other than the predicted one, a message is sent to resolve the inconsistency by setting it on the new correct path.

With reversible execution, the notification time is the time it takes for a peer's message to be passed to the other peers via the network:

$\tau_n = \tau_{erm}[a]$

where

$\tau_{erm}[a]$ = end-to-end reliable multicast delay from peer 'a' to all other peers

If the clocks on the various host computers involved in reversible execution are perfectly synchronized, the commit time for reversible execution is the earliest time of an unprocessed event or event still in transit on the network. Assuming events are processed as soon as they arrive at their destination host, the commit time is the maximum peer-to-peer reliable multicast delay between any two peers. This can be expressed as an inequality:

$\tau_c \leq \max(\tau_{erm}[an]) \; \forall \; n$

where

$\tau_{erm}[an]$ = end-to-end reliable multicast delay from peer 'a' to peer 'n'

This is the worst case where another event has been set in transit between a user's local peer and the peer with maximum reliable multicast delay with respect to it, just before your event was transmitted. This means that the local event will be rolled back (if it conflicts with the remote event).

The exact commit time is never calculated with reversible execution because it is too expensive in terms of bandwidth. However, because $t_c \leq \max(t_{erm}[an])$, it would be possible to keep track of the peer with the largest reliable multicast delay, thus obtaining an upper bound for the commit time.

For each update in reversible execution:

n_u	$= 1.r_m$
where	
n_u	= number of messages for an update

Discussion

Reversible execution provides optimal response, notification and requires significantly less bandwidth. This is particularly useful if users are going to change which objects they are interacting with frequently. Commit time is worse than the locking methods discussed, but is considered acceptable because it is assumed that most of the time conflicts will not occur.

Concurrency Control Summary

The above analysis shows that the choice of concurrency control mechanism affects the scalability and responsiveness of a multi-user virtual reality system. This section has a summary of the response times, notification times, commit times and bandwidth usage of the various mechanisms and compares the differences between them.

Response Time

Reversible Execution always has optimal (minimum) response time because operations are always reflected locally immediately. The other concurrency control mechanisms are pessimistic, and so have larger response times. Their operations are only reflected locally once a lock has been granted. Once the lock has been granted, operations can be reflected locally immediately. Their response times are thus equal to their commit times (see below).

In order to decrease the response times with these concurrency control mechanisms, they can try to predict when locks will occur and re-execute the simulation if they guess incorrectly. They then start take on the characteristics of a Reversible Execution mechanism such as the Replicated Time Warp mechanism. For an example of this type of work predictive work with locks, see Roberts et al (1995).

Commit Time

Concurrency Control	*Commit Time ('write' access needed), τ_{ci}*	*Commit Time ('write' access already granted), τ_c*
Client-Server Locking	$2.\tau_{eru}[as] + \tau_{sl}$	τ_{cs}
Peer-to-Peer Locking	$\max(\tau_{erm}[an] + \tau_{sln} + \tau_{eru}[na])$	τ_{cs}
Peer-to-Peer Master Entities	$2.\tau_{eru}[ap] + \tau_{st}$	τ_{cs}
Reversible Execution	earliest message still in transit $\leq \max(\tau_{erm}[an])$	earliest message still in transit $\leq \max(\tau_{erm}[an])$

Table 4 Concurrency Control Commit Times

The concurrency control schemes that use 'write' access have optimal commit times once the access has been granted (see third column in Table 4). This means that users performing operations using the pessimistic concurrency control mechanisms can manipulate their objects without the fear of their actions being altered. Other users, however, cannot perform operations on these objects while the lock is in place.

Reversible execution has a commit time equal to the earliest message still in transit. Users operating in a reversible execution environment will observe alteration of their actions if operations are performed concurrently (i.e. modifying the same object within a period equal to the commit time).

Comparing the various commit times when 'write' access needs to be obtained, it can be seen that client-server locking and peer-to-peer master entities have very similar formulae for commit times. Their commit times are proportional to the end-to-end delay between themselves and the host that currently possesses the lock. Peer-to-peer Locking is not far behind. Reversible execution, on average, will be the winner—although peer-to-peer master entities and client-server locking will have better performance for hosts close to the servers.

Notification Time

Concurrency Control	*Notification Time ('write' access needed), τ_{ni}*	*Notification Time ('write' access already granted), τ_{n}*
Client-Server Locking	$\tau_{eru}[as] + \tau_{si} + \tau_{sr} + \tau_{erm}[s]$	$\tau_{eru}[as] + \tau_{sr} + \tau_{erm}[s]$
Peer-to-Peer Locking	$\max(\tau_{erm}[an] + \tau_{sin} + \tau_{eru}[na]) + \tau_{erm}[a]$	$\tau_{erm}[a]$
Peer-to-Peer Master Entities	$(2.\tau_{eru}[ap] + \tau_{st}) + \tau_{erm}[a]$	$\tau_{erm}[a]$
Reversible Execution	$\tau_{erm}[a]$	$\tau_{erm}[a]$

Table 5 Concurrency Control Notification Times

The client-server architecture is the odd one out when it comes to the notification time once 'write' access has been granted. It will always have a slightly larger notification time because the server must relay the message. In a star topology, the time will be very similar to the times of the other schemes:

$$\tau_{eru}[as] + \tau_{sr} + \tau_{erm}[sb] \approx \tau_{erm}[ab]$$

This is because the end-to-end delay advantage of reliable multicasting is less since the server (at the centre of the star) is the nearest peer.

Bandwidth

Concurrency Control	*Number of messages to obtain 'write' access, n_i*	*Number of messages to perform an update, n_u*
Client-Server Locking	$3.u_r$	$1.u_r + 1.m_r$
Peer-to-Peer Locking	$2.m_r + (n_p\text{-}1).u_r$	$1.m_r$
Peer-to-Peer Master Entities	$2.u_r$	$1.m_r$
Reversible Execution	0	$1.m_r$

Table 6 Concurrency Control Numbers of Messages

Client-server has a disadvantage when it comes to bandwidth because each object update requires one reliable unicast and one reliable multicast message. The other concurrency control schemes require only one reliable multicast.

When it comes to obtaining 'write' access, reversible execution has the upper hand since it is not required to obtain 'write' access, thus it does need to send any messages. Peer-to-peer master entities is next, requiring two unicasts. Client-server locking requires three unicasts. Peer-to-peer locking requires the largest bandwidth (assuming there is more than one other peer) with a minimum of two reliable multicasts plus a reliable unicast from each peer.

Conclusion

The performance of concurrency control schemes is, to a certain extent, applications dependent. Schemes where 'write' access is required (client-server locking, peer-to-peer locking, peer-to-peer master entities) perform very well once a lock has been acquired, but other users cannot modify the locked objects. They do require more bandwidth and suffer lower commit and notification times when locks need to be obtained. For these reasons, locking schemes perform best when:

- Users rarely change which objects they are manipulating, and other users do not mind not being able to manipulate the object while someone else is.
- Critical operations are performed that should not be rolled back.
- Bandwidth is available if users wish to change which objects they are manipulating frequently.

Reversible execution does have the best notification times and requires the least bandwidth, but it does not have the best commit times. For this reason, reversible execution performs best when:

- Users change which objects they are manipulating frequently, or wish to modify an object that is frequently accessed by other users.
- Operations performed can be rolled back without serious user interface effects.
- Operations, more often than not, do not conflict and cause rollbacks.
- Bandwidth is limited.

A locking scheme could be merged with a reversible execution scheme to provide more flexibility for virtual world designers and better performance for the end users. Locking provides better commit times than reversible execution, once a lock has been granted. It would be nice if the designer could choose which scheme to use for different objects.

2.5 Summary

This chapter has discussed various design issues for multi-user virtual reality systems. These included message passing techniques, remote entity approximation, data distribution and concurrency control. These techniques can be combined to produce efficient multi-user virtual reality systems that can solve scalability, responsiveness

and consistency issues. This information can now be used to compare various multi-user virtual reality systems in the Related Work section.

3 RELATED WORK

This chapter describes various multi-user virtual reality systems that exist today. Each system uses one or more of the techniques described in the previous chapter to meet its requirements for data distribution, concurrency control and scalability.

3.1 SIMNET

The SIMulator NETwork (SIMNET) was developed for group training exercises and implemented between the years of 1983 and 1990. The project was sponsored by the then Defense Advanced Research Projects Agency (DARPA, now called ARPA) and the United States Army.

SIMNET supported ground based vehicles such as tanks, and air based vehicles such as aeroplanes. These could be manned vehicles or Semi-Automated Forces (SAFs). SAFs were implemented just like manned simulators, except that a computer controlled the actions of the objects. Usually a human 'commander' would supply instructions to the SAFs during the simulation to further direct their actions.

Applications of SIMNET included (Miller & Thorpe 1995),

- Forward Area Air Defense System, 1988-1989 — Weapon prototyping.
- Combat Vehicle Command and Control, 1988 — Equipment prototyping.
- Nonline-of-Sight Missile, 1989-1990 — Weapon prototyping.
- Counter Target Acquisition System, 1990-1991 — Weapon prototyping.
- Line-of-Sight Anti-Tank Missile, 1990 — Weapon prototyping.
- Battle Reenactment — A section of the Gulf War was recreated using SIMNET's Semi-Automated Forces. Hailed by military historians as a benchmark for battle recreation.

The major downfalls of SIMNET were that it used expensive specialized hardware available from only one vendor for the user interface and custom network code so that it could only run over certain networks. Figure 27 shows a typical SIMNET network setup.

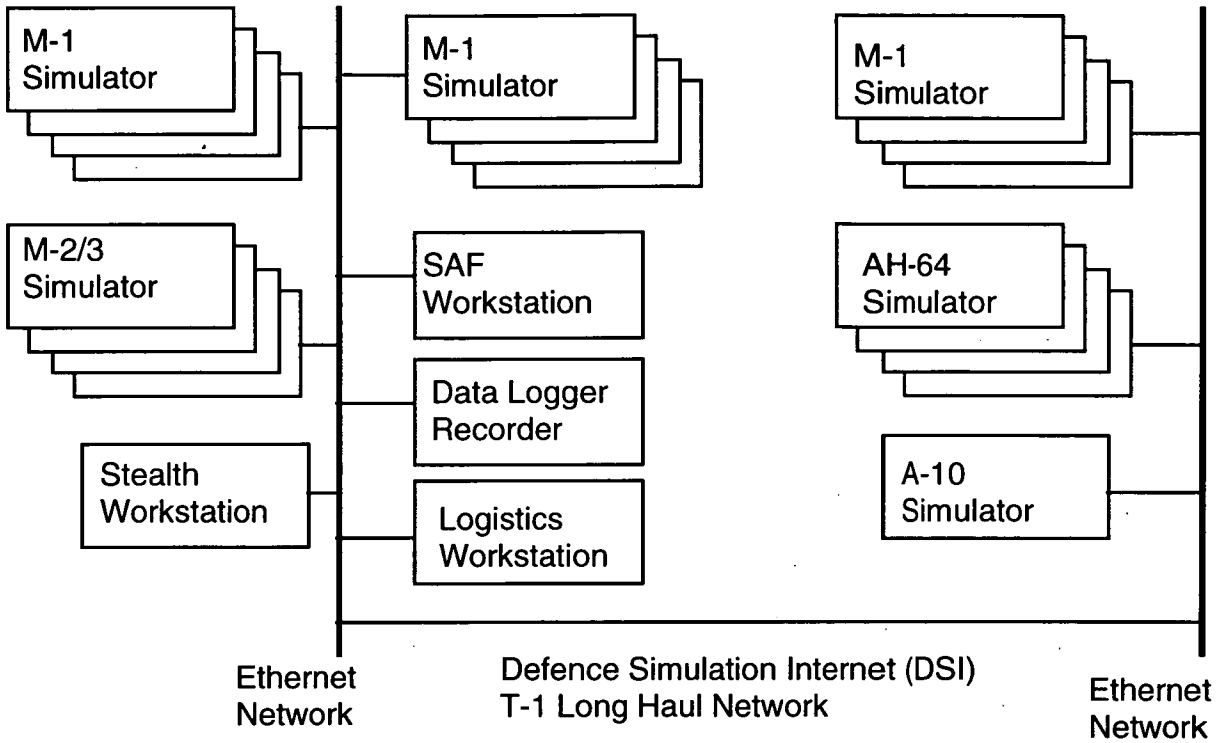


Figure 27 Typical SIMNET Network Setup (Pratt 1993)

SIMNET used dead-reckoning and broadcasting for position and orientation updates. Consequently, local object position and orientation updates were immediately executed at the local host, providing optimal response time. No concurrency control was needed for position and orientation updates because they were only modified by the local host, so no concurrent modification occurred. Notification times were optimal for position and orientation updates as well because of this reason.

Timestamp information was required on each event so that simulations can be recorded and played back for later analysis. The timestamp could also be used to compensate for latency variance.

As a rule of thumb for SIMNET simulations, total latencies should not have exceeded typical human reaction times (250ms) or anomalies in causality could become apparent (Miller & Thorpe 1995).

The limiting factor in SIMNET was not network bandwidth, but host processing performance (Macedonia 1995). SIMNET, however, ran over a dedicated, high-speed network. This is generally not available. We want to develop a system that will allow other activities on the network as well (such as other running virtual worlds).

Data Distribution

One of the aims of SIMNET, due to its military applications, was to eliminate the need for any centralized control—the same design goal that the Internet had. A peer-to-peer data distribution mechanism was thus used due to its increased fault tolerance.

In order for all the hosts to remain consistent throughout a simulation, at the beginning of each simulation every host must somehow already have a replica of the terrain database. The terrain database could not be modified in SIMNET since there was no mechanism for communicating the changes to other peers. The dead-reckoning algorithms that were going to be used would also be known at the beginning of the simulation for consistency reasons.

Hosts could join and leave simulations dynamically (while the simulation was running). This was possible because each entity was required to send *absolute* updates about itself periodically (typically every 5 seconds). A complete picture of the simulation can thus be built within that period (assuming the initial terrain database and dead-reckoning algorithms are already available).

Concurrency Control

Each host was responsible for one or more entities. The host would send updates for the entities it handled and any events they caused. Interaction between entities was limited. The most common interactions were entities shooting each other using various weapons. Entities receiving events (entities being shot at) were responsible for calculating the effect the event had on itself (responsible for assessing its own damage) and for notifying others of changes to itself. This was essentially a peer-to-peer master entity concurrency control mechanism, although there was no built in provision for migration and the master entity was maintained by the same host throughout the simulation. This mechanism worked very well in this system because objects were typically evenly distributed amongst the hosts, usually one per host, so network loads and computational loads were evenly distributed.

Even though unreliable UDP messages were used, reliability is ensured because messages are always constantly updated as 'ground truth' (absolute rather than relative). This means that even if messages are missed, if a later packet is received, it will contain the up-to-date information. As an example, when an entity is destroyed, the fact that it is dead is repeated every five seconds or so, so that hosts that missed the initial message or new hosts joining the simulation are up-to-date.

Summary

Design Issue	Implementation
Message Passing	Broadcasting
Remote Entity Approximation	Dead-Reckoning
Data Distribution	Peer-to-Peer
Concurrency control	Peer-to-Peer Master Entity

Developer

Defense Advanced Research Projects Agency and the United States Army

References

Locke 1994

Miller & Thorpe 1995

3.2 DIS

The Distributed Interactive Simulation (DIS) standards are a group of IEEE standards (IEEE standard number 1287) that address communications architecture, format and content of data, entity information and interaction, simulation management, performance measures, radio communications, emissions, field instrumentation, security, database formats, fidelity, exercise control and feedback (Macedonia et al 1994). It is based on the principles of SIMNET. Development began in 1989 and the first standard was approved in 1993.

Significant changes from SIMNET include (Miller & Thorpe 1995),

- A change in the coordinate system from a rectangular Cartesian coordinate system representing a flat earth to a Cartesian system with the origin at the centre of the earth that can represent a curved earth.
- Various networking requirements are stated. End-to-end delay should be less than 300ms for 'loosely coupled' interactions (such as observing entities at a distance) and 100ms for 'tightly coupled' interactions (such as flying in formation). There should also be low latency variance and reasonably reliable delivery (around 1-2% randomly distributed packet loss).

The DIS standard has a great advantage over SIMNET because it can use 'off the shelf' equipment, rather than expensive custom-built hardware.

There is a lot of ongoing work aimed at improving the DIS standard. Several major groups are developing different sections of the protocol. Annual workshops are held to discuss improvements and are attended by hundreds of people. One example of an area of research is that of terrain databases. These have to be consistent among various simulators, but differences in internal representations of these databases have meant that a standard was not agreed upon for the first DIS standard.

A new generation of DIS standards is under development that will enable far greater numbers of entities to participate in the simulations. The new DIS will have capabilities such as multicasting and will have smaller message sizes.

Data Distribution

DIS requires the virtual world to be stateless (Macedonia et al 1995). The state of the entire world can be obtained simply by listening to state information updates from all the objects. This simplifies the system, but also means that redundant information is sent about objects. Hosts can, however, join the simulation easily at any stage. This is inefficient when a significant number of static entities exist because updates to entities that are known not to change much during the simulation need only be done once. Examples of static entities include various bits of terrain that can be destroyed, and destroyed vehicles. Examples of static information that is transmitted redundantly include identifying markings on vehicles, which would typically be the same throughout the simulation. These also need to be sent each time that an object update is sent.

Concurrency Control

See SIMNET.

Broadcasting is still used for entity updates because the entities in DIS simulations are typically densely distributed (Macedonia et al 1995), so partitioning and multicasting would have limited benefit. This stems from the origins of SIMNET and DIS, where they were used mainly for smaller scale simulations on LANs.

Summary

Design Issue	*Implementation*
Message Passing	Broadcasting
Remote Entity Approximation	Dead-Reckoning
Data Distribution	Peer-to-Peer
Concurrency control	Peer-to-Peer Master Entity

Developer

A joint effort by government/industry groups, supported by the US Army Simulation, Training and Instrumentation Command (STRICOM), the Defence Modeling and Simulation Office, and (initially) the Advanced Research Projects Agency (ARPA).

References

Davis 1995

Fitzsimmons & Fletcher 1995

Hofer & Loper 1995

Miller & Thorpe 1995

Pullen & Garrett 1995

Reddy & Wood 1995

Shiflett et al 1995

Stytz 1996

3.3 NPSNET IV

NPSNET IV aimed at increasing the number of entities that could be supported by a military simulation to well over 1000. This was realized by using multicasting (as opposed to broadcasting). As a result, NPSNET IV became the first system to use both the DIS standard and multicasting.

Figure 28 shows the history of NPSNET I to IV:

- NPSNET I & II used a locally designed network scheme that required Ethernet as the LAN protocol and used an ASCII-encoded application level protocol to convey simulation state. Consequently, NPSNET I & II were restricted to Ethernet LANs and had large network message lengths.
- NPSStealth was a version of NPSNET that had a translator for the SIMNET protocol that enabled NPSNET participants to interact with SIMNET simulations over LANs and WANs.
- NPSNET IV adopted the DIS protocol for interoperability and thus created a more scalable software and network architecture.

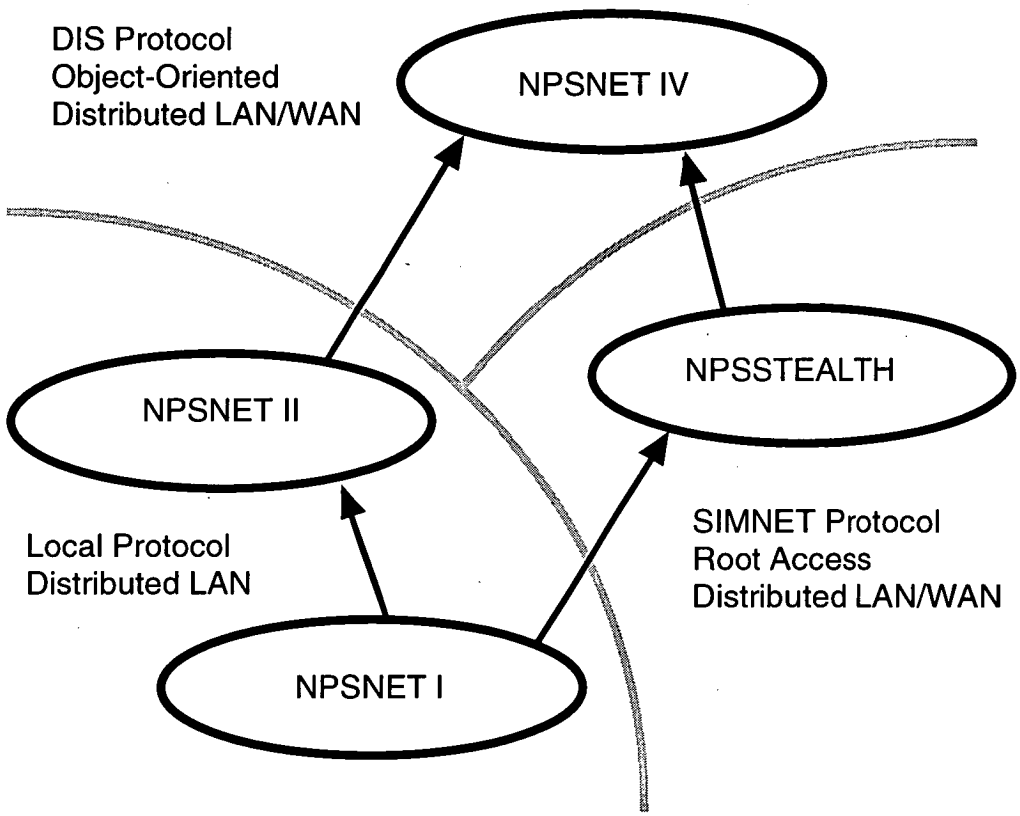


Figure 28 Evolution of NPSNET Networking (Zeswitz 1993)

Data Distribution

Although NPSNET IV's architecture is targeted at military simulations, its principles for partitioning can be applied to other multi-user virtual reality systems:

NPSNET IV takes advantage of the fact that not all entities in a virtual world need to know about each other at the same time. It does this by partitioning the entities in its virtual world into groups (or classes) based on their particular interests. Each group is

associated with its own multicast group, thus each group is essentially independent of the other. NPSNET IV supports functional, spatial and temporal groups.

Spatial Groups

The creators of NPSNET IV observed that in real military exercises, vehicles did not move much. For example, during a ten hour exercise one third of vehicles did not move and as the simulation progressed, over half the vehicles became disabled and thus stopped movement. In addition, 60% of the terrain was outside the detection range of all vehicles. A simulated infantryman, for instance, in a virtual world does not need to know the condition of a simulated truck 20 kilometres away. Other studies have found that land combat operations stand still 90 to 99% of the time. The world record for aggregate movement in modern warfare was 92 kilometres per day for four days (about 6 kilometres per hour) in Desert Storm. Individual vehicles move much faster but would not continue at high rates for long because they fight as part of units whose movement must be coordinated (Macedonia 1995 et al).

In NPSNET IV the virtual world's surface is partitioned into a honeycomb of hexagons. Each hexagon has its own multicast address and entities can subscribe to partitions they are interested in. Entities can belong to more than one group at a time to avoid boundary or temporal aliasing (see Figure 29).

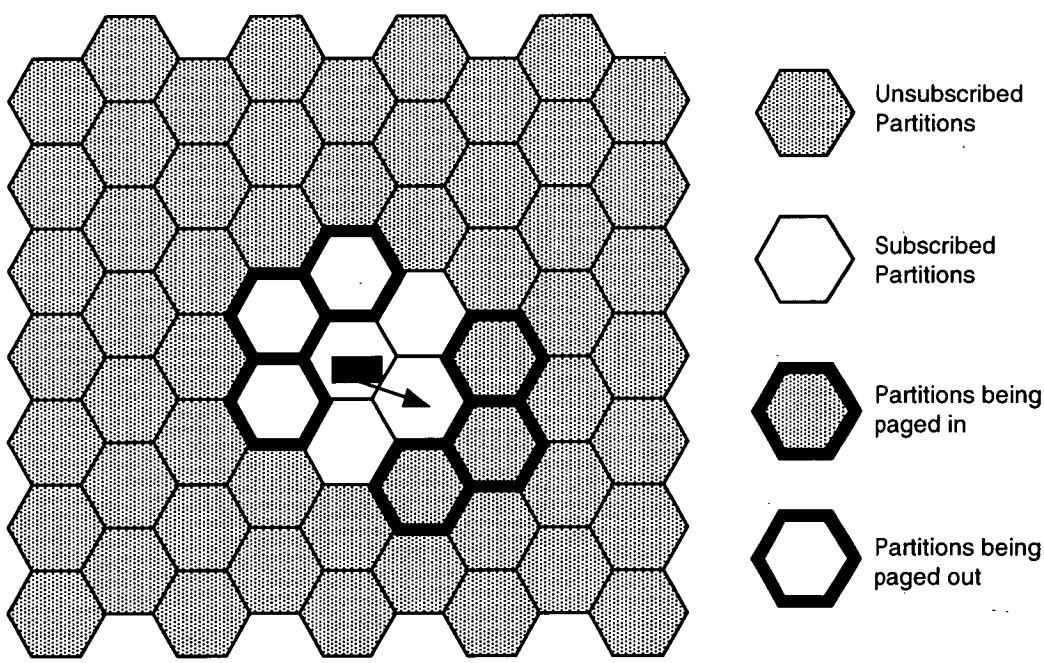


Figure 29 Spatial Partitioning in NPSNET IV (Macedonia et al 1995)

Functional Groups

Functional partitioning involves dividing members up according to their needs based on functionality. Radio communication messages, for instance, would be sent only to those entities that can receive them.

Temporal Groups

Temporal partitioning involves having groups that receive virtual world state updates at different rates, for example, once per second or once per minute. Certain entities require faster updates than others do. A space-borne sensor, or a system management entity might only need low-resolution information, thus infrequent updates. These entities need only subscribe to temporal groups that supply the minimum rate they are interested in (see Figure 30).

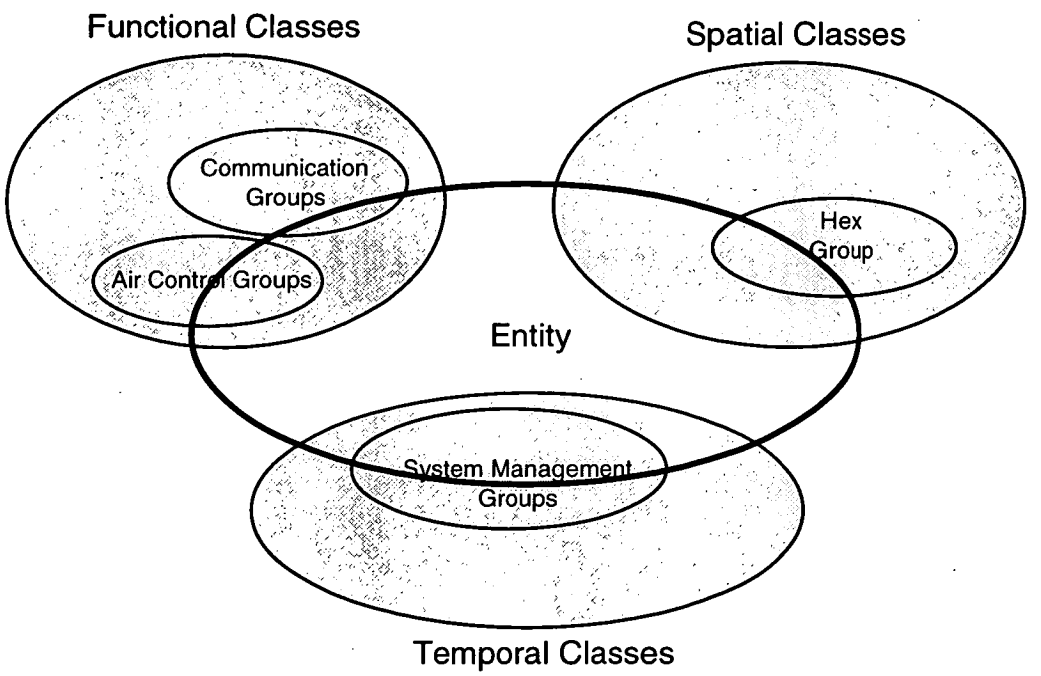


Figure 30 One Entity may Belong to Several Spatial, Functional and Temporal Groups (Macedonia et al 1995)

Concurrency Control

Unchanged from SIMNET and the DIS standard.

Summary

Design Issue	*Implementation*
Message Passing	Multicasting
Remote Entity Approximation	Dead-Reckoning
Data Distribution	Peer-to-Peer with Partitioning
Concurrency control	Peer-to-Peer Master Entity

Developer

Naval Postgraduate School, Monterey, California

References

http://www.npsnet.nps.navy.mil/npsnet

Macedonia et al 1994

Macedonia et al 1995

3.4 HLA

The High Level Architecture (HLA) is an important step in standardising distributed simulations. In 1996 it was declared that the High Level Architecture (HLA) be the standard for all US Department of Defence simulations. After 1999, no more funding will be given to non-HLA simulations, and by 2001 all non-HLA simulations will be phased out.

The HLA facilitates interoperability and re-use amongst simulation developers, which, in turn, gives greater capability and cost-effectiveness. Figure 31 shows the basic architectural components of the HLA. It consists of: the Runtime Infrastructure (RTI), the Interface, and the Simulations.

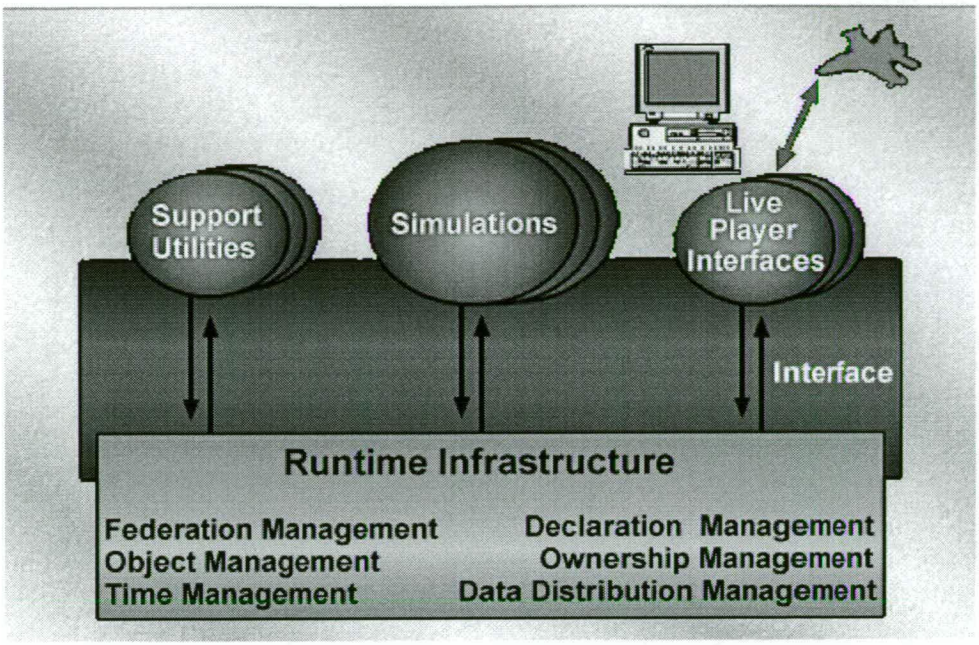


Figure 31 HLA Architectural Components (HLA Training CD 1999)

In the HLA, the Simulations are defined as consisting of Federations and Federates. The Federations are classes of simulations, and Federates are the classes of entities within those simulations. They are defined more formally as follows:

- *Federation* — A set of interacting federates, a common federation object model and supporting RTI that are used as a whole to achieve some specific objective.
- *Federates* — Members of an HLA Federation. They have a Simulation Object Model (SOM), documented in accordance with the HLA Object Model Template (OMT).
  - *Object Model Template (OMT)* —The common method for recording the information contained in the required HLA Object Model for each federation and simulation. It fosters interoperability and reuse of simulations via the specification of a common representational framework.
  - *Simulation Object Model (SOM)* — Describes objects, attributes and interactions in a particular simulation that can be used externally in a federation.
  - *Federation Object Model (FOM)* — Describes all shared information (objects, attributes, interactions and parameters) essential to a particular federation.

The two other components of Figure 31 are the Interface and the Runtime Infrastructure:

- *Interface* — Specifies the interface between the Federates and the Runtime Infrastructure.
- *Runtime Infrastructure* — Provides the following services via the interface to the Simulations:

- *Federation Management* — Create and delete federation executions. Join and resign federation executions. Control checkpoint, pause, resume, and restart.
- *Declaration Management* — Establish intent to publish and subscribe to object attributes and interactions.
- *Object Management* — Create and delete object instances. Control attribute and interaction publication. Create and delete object reflections.
- *Ownership Management* — Transfer ownership of object attributes.
- *Time Management* — Coordinate the advance of logical time and its relationship to real time.
- *Data Distribution Management* — Support efficient routing of data.

Figure 32 shows the usage of the various RTI components during a simulation's lifecycle. Federation management is the first service to be used. Ownership management is only used during the operation of the simulation, after the other housekeeping issues have been dealt with.

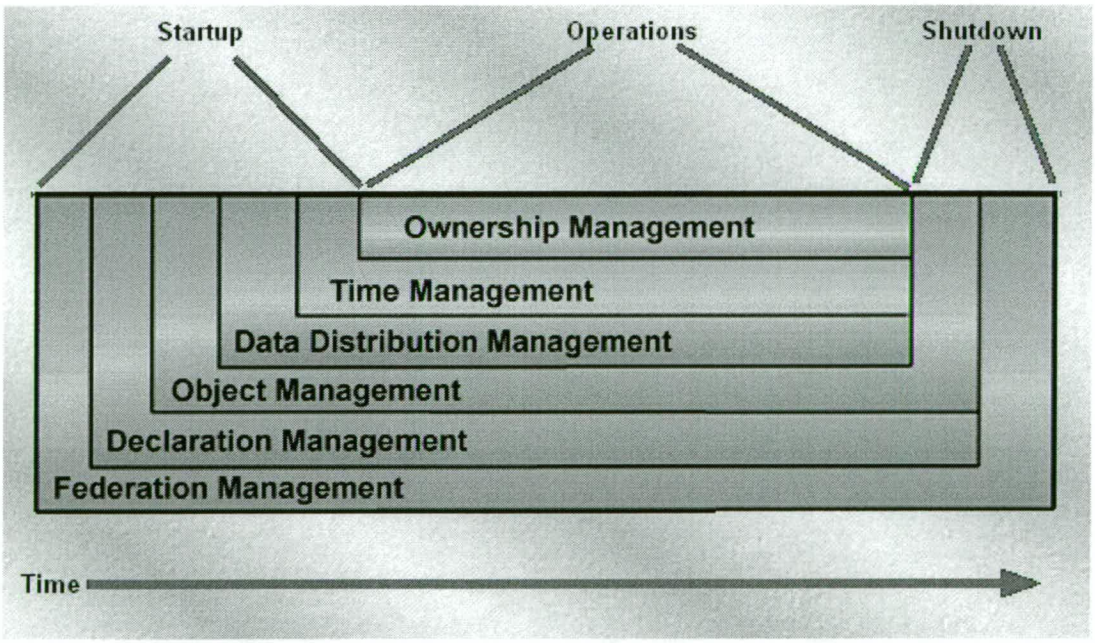


Figure 32 HLA RTI Services Over the Life of a Federation (HLA Training CD 1999)

Several RTI implementations are available with API bindings for most languages. Some implementations are available free of charge, and can be downloaded from the Internet.

Data Distribution

The following RTI services are available to aid efficient data distribution:

- *Object Management* — provides the basic data distribution facilities. It enables creation, destruction, and replication of objects. It can be used to specify whether object replica updates are sent reliably or unreliably.
- *Declaration Management* — Each object is an instance of an object class found in the FOM. Each object class has a set of attributes associated with it. Federates use Declaration Management to declare intention to publish attributes and subscribe to attributes from an object class. Object classes are chosen by the simulation designer to facilitate a desired organisational scheme. As an example, two classes of objects may be provided: Land-Based Vehicles (subclasses might include truck, motorbike), and Air-Based Vehicles (subclasses might include helicopters and jets). An instance of a motorbike federate can then subscribe to the Land-Based Vehicles class to be sent only information about land-based vehicles. This provides functional partitioning (see NPSNET IV).
- *Data Distribution Management* — provides partitioning. Regions can be defined for a federation. Object replicas will only receive updates sent to the regions in which they are located. Data Distribution Management can thus be used for spatial partitioning. The axes of the space in which these regions are located (and the number of axes) is user defined, and they do not necessarily have to be spatial dimensions. An example of a useful dimension would be 'radio frequency'. If object replicas are inside a 'region' of the correct radio frequency, they will receive radio messages published to that frequency.

Data Distribution Management and Declaration Management provide partitioning. They way the RTI implements the distribution of object replica updates are implementation dependent. Ideally, hardware multicasting would be used.

Federates can send attribute updates at their own rate, therefore have the ability to implement dead-reckoning or other similar bandwidth reducing techniques. The conditions applicable to the update of specific instance attributes of a federate are documented in the SOM for that federate.

Concurrency Control

The following RTI services are available to aid concurrency control:

- Ownership Management

The HLA uses Peer-to-Peer Master Entities for concurrency control. It provides the facility for it to be finer grained than just a per-object basis. Individual attributes can be owned by different federations. Only the federate that owns an instance attribute can update the attribute. There is a special "privilegeToDeleteObject" attribute that needs to be owned by a federation in order to delete an object from it. The RTI Ownership

Management service allows ownership of attributes to be transferred at runtime between Federations.

Summary

Design Issue	*Implementation*
Message Passing	RTI Specific
Remote Entity Approximation	Federate Specific
Data Distribution	RTI Specific, with provision for Partitioning
Concurrency control	Peer-to-Peer Master Entity

Developer

DMSO, US DoD

References

IEEE P1516/D1, Draft Standard [for] Modeling and Simulation (M&S), High Level Architecture (HLA) – Framework and Rules, HLA Training CD 1999

IEEE P1516.1, M&S HLA - Federate I/F Spec, DRAFT 1, HLA Training CD 1999

http://hla.dmso.mil/

3.5 DIVE

DIVE is a UNIX based application that runs over the Internet.

Objects with behaviours in DIVE are called *actors*. An actor is a process that can interact with the virtual world. It is either autonomous or user controlled. Actors can enter different virtual worlds by passing through *gateway objects*. When an actor collides with a gateway object, the multicast address of the new virtual world is obtained from a name server and the actor is transferred to the new virtual world. DIVE actors can communicate using text messages or live audio.

Objects can have behaviours in DIVE. These are in the form of DIVE/Tcl scripts, which gives them the ability to be transmitted over the network and executed on any platform immediately (without compilation). Scripts are usually activated as the result of an event.

Data Distribution

DIVE is a peer-to-peer system that implements partitioning by supporting multiple independent virtual worlds. Each virtual world is assigned a unique multicast address that is used for communication amongst members of the particular virtual world. This ensures that only messages that a certain host is interested in are received by it, thereby reducing processing load and required network bandwidth. A DIVE name server handles mapping virtual world names onto the multicast addresses.

If a DIVE peer wants to join a virtual world, it contacts a name server to get the multicast address of the world. The first peer that joins the world obtains a complete description of it from a server. Each peer that connects after that obtains the (possibly changed) world description from its closest peer.

DIVE uses a NAK reliable multicast protocol for message passing. If a peer detects a missing update to an object, it requests the latest state of the object from (in the ideal case) the closest peer. This eliminates the need for the multicast algorithm to store a list of sent messages just in case one needs to be re-sent.

Concurrency Control

It is assumed that actors own objects for a long time and that concurrent modifications seldom occur. Under these assumptions, it is not surprising to see that peer-to-peer master entity concurrency control is used. Actors own objects and can then write to them. The ownership can be passed. In the case of conflict, one actor blocks until it receives ownership.

Earlier versions of DIVE used positive acknowledgment multicasting, peer-to-peer distributed locking and no partitioning, but did not manage to scale past 10 peers on a LAN. Newer versions of DIVE use NAK reliable multicasting, peer-to-peer master entity concurrency control and dead-reckoning to support 20 participants on a WAN when latencies are less than 200ms.

Summary

Design Issue	*Implementation*
Message Passing	Reliable Multicasting
Remote Entity Approximation	Dead-Reckoning
Data Distribution	Peer-to-Peer with Partitioning
Concurrency control	Peer-to-Peer Master Entity

Developer

Swedish Institute of Computer Science.

References

http://www.sics.se/dive

Carlsson & Hagsand 1993

Hagsand 1996

3.6 SPLINE

SPLINE (Scalable PLatform for Interactive Environments) is a middleware layer that is used by applications for the development of multi-user virtual worlds at Mitsubishi Electric Research Laboratories (see Figure 33). A medium sized virtual world known as 'Diamond Park' has been successfully implemented using SPLINE. Efforts are being made to make SPLINE run efficiently over high-speed ATM networks.

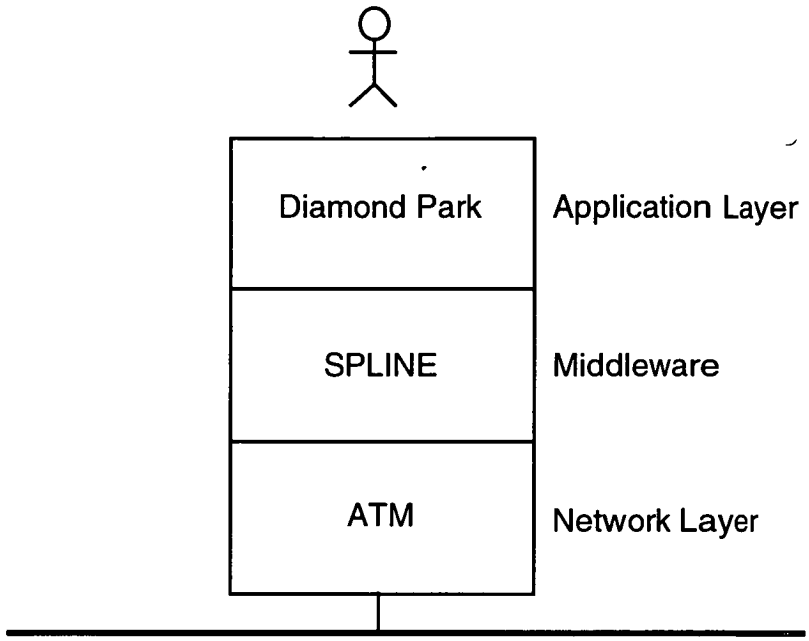


Figure 33 SPLINE — a Middleware Layer

SPLINE is aiming to be an open system where large numbers of people can easily develop content for the virtual worlds. This will mean that the content of the environment can grow in proportion to the talent of the user community, not just the system implementers.

Data Distribution

SPLINE is a peer-to-peer system that supports dead-reckoning, multicasting and partitioning.

Virtual worlds in SPLINE are partitioned into arbitrarily shaped 'locales'. Each locale has its own floating-point coordinate system that gives high precision at the centre of the locale, decreasing with distance. There is no global coordinate system. Locale positions are relative to each other.

Locales provide efficient location-addressable communication. The partitioning into locales provides a degree of encapsulation that aids in the integration of independently designed locales into one virtual world.

Beacons provide content-addressable communication. Beacon objects can be created that regularly send information about themselves to a multicast address calculated by hashing the beacon's tag onto a set of multicast addresses. When a host is interested in a particular beacon object and knows its tag, it can listen to the appropriate multicast address for information about that object.

Concurrency Control

SPLINE uses peer-to-peer master entity concurrency control. Objects created have ownership and only the owner can modify the object. Ownership can be transferred.

Summary

Design Issue	*Implementation*
Message Passing	Multicasting
Remote Entity Approximation	Dead-Reckoning
Data Distribution	Peer-to-Peer with Partitioning
Concurrency control	Peer-to-Peer Master Entity

Developer

Mitsubishi Electric Research Laboratories, 201 Broadway, Cambridge, Massachusetts 02139

References

http://www.merl.com/

Anderson et al 1995

Barrus et al 1996

Walters 1996

Walters et al 1996

3.7 BrickNet

BrickNet is a high-level toolkit for creating multi-user virtual worlds that runs on Silicon Graphics workstations over the Internet.

BrickNet uses a client-server architecture. Each client executes its own virtual world. Users can interact with the client virtual world via the Interaction Support Layer. Objects within the virtual world can either be local to the client or shared. Updates to shared objects are sent via a server.

A Client supports a number of layers (see Figure 34):

- *Interaction Support Layer* — Enables the interaction of the user with the virtual world.
- *VR Knowledge Layer* — Simulates the virtual world (i.e. calculates object behaviours).

Server layers include (see Figure 34):

- *Client Management Layer* — Handles client specific data such as the joining and leaving of clients from the server.
- *Object Management Layer* — Handles updates to objects and the associated concurrency control mechanism.
- *Update Request Layer* — Ensures consistency among clients by receiving updates to objects and (access permission allowing) processing and sending the updates to all other interested clients.
- *Communication Layer* — Implements the sending of messages between clients and servers. It uses UDP for communication.

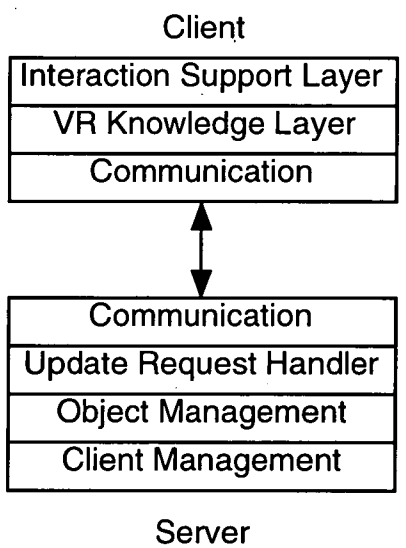


Figure 34 Architecture of BrickNet Clients and Servers.

Behaviours can be transmitted as part of an object over the network. The code is in 'Starship' format, an interpreted language. When an object's behaviour is transmitted, the source code can simply be transmitted.

Data Distribution

Clients connect to a server in order to receive and send data about shared objects. The servers communicate with each other to distribute the information amongst themselves (see Figure 35).

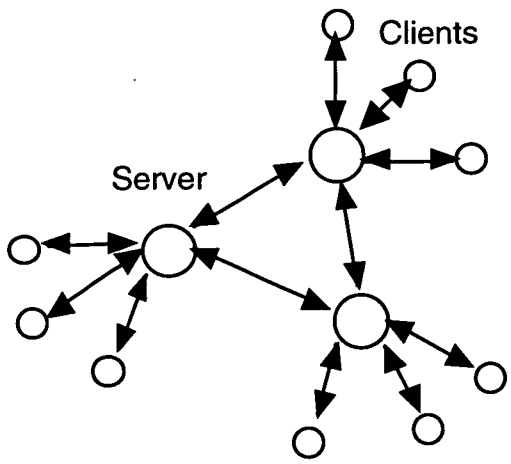


Figure 35 BrickNet Client-Server Interactions

BrickNet allows users to have both shared and private workspaces and objects. Shared workspaces in BrickNet use replication on demand whereby information regarding shared objects is sent only to other clients that explicitly express interest in them. This is a simple form of partitioning.

Unsynchronised behaviours are supported where the object executes its behaviour independently of the other clients (these are not particularly interesting in a multi-user virtual reality context).

Synchronized behaviours are also supported. Synchronized behaviours can be used to implement dead-reckoning, and other more complex behaviours. Synchronized behaviours are implemented such that a client can execute an object's behaviour at a local client and be periodically synchronized by update messages. Behaviour code can also be distributed dynamically (while the virtual world is running).

Concurrency Control

A client can choose to share an object it owns. Remote clients can then 'lease' that object. When the owning client wants to synchronize the object at the different clients, it sends a sync message, containing relevant synchronization data, to the server, which then passes the message to the appropriate clients. Reliable communication is ensured by the server: It transmits the sync message to the clients that have 'leased' the object, and receives acknowledgments from them. Once all clients have sent

acknowledgments to the server, the server notifies the original client that the synchronization was successful. BrickNet's client-server locking scheme is limited because ownership of objects cannot be transferred.

Summary

Design Issue	*Implementation*
Message Passing	Unicasting
Remote Entity Approximation	Locally Simulated Objects with Arbitrary Behaviours
Data Distribution	Client-Server
Concurrency control	Client-Server Locking

Developer

Institute of Systems Science, National University of Singapore, Singapore.

References

Singh et al 1995

3.8 Environment Manager

The Environment Manager (EM) is a high level tool for constructing single or multi-user virtual worlds. EM is built on top of the Minimal Reality (MR) Toolkit Peer Package, which provides the user interface and Internet networking capabilities (using UDP).

Behaviours are supported: a client can execute an object's behaviour at a local client and be periodically synchronized by update messages. Objects and their behaviours in an EM virtual world are described by the Object Modeling Language (OML), an interpreted C-like programming language. EM supports multi-user-different-content as well as multi-user-same-content. This is similar to the synchronized and unsynchronised behaviours available in BrickNet.

Data Distribution

EM is a peer-to-peer system. The MR Toolkit's Peer Package that EM relies on for communication does not support hardware multicasting. The MR Toolkit performs software multicasting by sending unicast messages to interested peers. Future versions of the MR Toolkit may provide a hardware multicasting facility.

Concurrency Control

Objects can have shared instance variables for sharing themselves amongst peers. These variables can be assigned either 'writeable' or 'readable' permissions.

Writeable permission is essentially a way of bypassing the concurrency control mechanism, allowing any peer to change instance variables with this permission at any time. This permission is useful for interactions that will not affect the long-term consistency of the virtual world, such as object position and orientation updates. The increased responsiveness of writeable permission over readable permission outweighs the possible inconsistencies caused.

Readable permission invokes a peer-to-peer master entity concurrency control mechanism on the specified instance variable. This is implemented using ownership token passing. Only the owner can write to 'readable' instance variables. Inconsistencies cannot occur because only the owner has write permission. The owner will send the current value to interested peers if necessary.

Summary

Design Issue	*Implementation*
Message Passing	Unicasting
Remote Entity Approximation	Locally Simulated Objects with Arbitrary Behaviours
Data Distribution	Peer-to-Peer
Concurrency control	Peer-to-Peer Master Entity

Developer

Department of Computing Science, University of Alberta, Canada.

References

http://www.cs.ualberta.ca/~graphics

Wang 1994

Wang et al 1995

3.9 Summary

This chapter has described various popular multi-user virtual reality systems that exist today. Each system uses some of the techniques described in the previous chapter to help meet its requirements for scalability, responsiveness and consistency. These systems provide a reference point for analysing the Replicated Time Warp that is presented in the next chapter.

4 REPLICATED TIME WARP

This chapter presents the main original contribution of this thesis, the Replicated Time Warp (RTW). The RTW is a concurrency control mechanism designed to be efficient at maintaining consistency between remote virtual worlds in multi-user virtual reality systems. It can be classified as a Reversible Execution concurrency control mechanism (see Design Issues, Concurrency Control), but the RTW adds some unique features: It allows *time dependent* deterministic objects to be simulated at a local host without sending any synchronization information, while still maintaining 100% consistency between remote virtual worlds. The only information that needs to be transmitted is non-deterministic interactions with these objects from the users. RTW is more suited to simulations that have high numbers of deterministic objects and lower numbers of non-deterministic interactions.

A perfect example of where the RTW will excel can be constructed: Consider a 'Gas Molecule Simulation' that can be interacted with. The simulation consists of many small molecules that bounce around in a container. Molecules also interact with each other by bouncing off one another.

If we use the RTW, no synchronization traffic for the deterministic molecule simulation is required in order to keep the simulations consistent. Only non-deterministic interactions by users with the molecules need to be transmitted. If we used any other method, traffic proportional to the number of simulated molecules would be required in order to keep simulations consistent when non-deterministic interactions with the molecules occurred (see Figure 36).

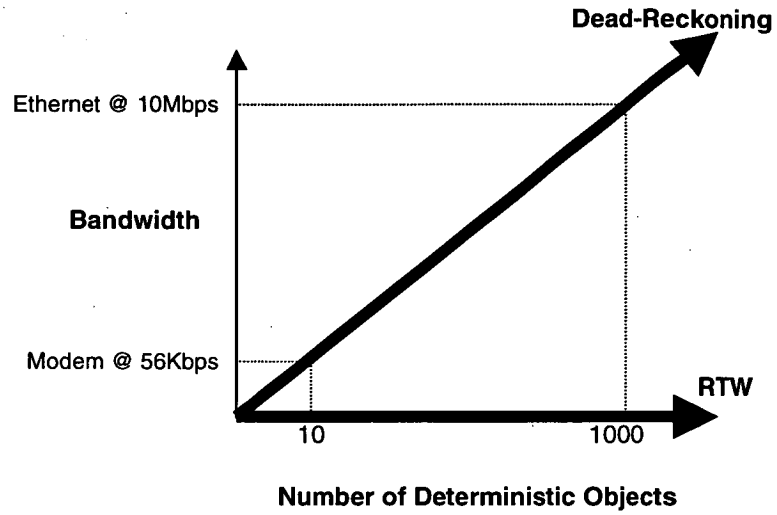


Figure 36 Bandwidth versus Number of Deterministic Objects

As the number of deterministic objects increases, RTW requires proportionally less total bandwidth than any other system. The following section explains how the synchronization process achieves this.

Eliminating Network Traffic caused by Deterministic Objects

Virtual worlds include *deterministic objects*. Deterministic objects have completely predictable and repeatable behaviour[1]. Due to this predictability, there is scope for reducing synchronization network traffic caused by them, or even eliminating it. Even seemingly random behaviours can be introduced in a deterministic manner using pseudo random number generation.

No communication need occur between virtual worlds if all the objects in the virtual world are deterministic and the virtual worlds are exact replicas of each other. Remotely located users will observe exactly the same virtual world evolving at the same rate if they initially synchronize their local real-time clocks and the simulations stay in sync with their local real-time clocks.

When non-deterministic events affect a deterministic virtual world it is possible to maintain consistency among replicas if each event is introduced consistently into each virtual world. This means introducing the event at the same virtual time at each replica. Time is important because each virtual world will be dynamically evolving, so an event will have different effects depending upon when it is introduced.

Figure 37 illustrates this time dependency by considering a virtual world consisting of a deterministic 'puck' object bouncing around in a box. The two virtual worlds in the

[1] In this case deterministic means deterministic with respect to the RTW system. That is, the RTW system can predict the behaviours of these objects. Any object whose behaviour relies on events external to the RTW system are classified as non-deterministic with respect to the RTW system even though they may be deterministic in their original context.

figure are replicas of each other and are connected via a network. At a certain point in time, a non-deterministic event is inserted into the virtual world that causes the box to be partitioned into two halves. In the virtual world where the non-deterministic event originated, the puck is trapped in the left-hand side of the box. Due to the speed of light limitation, the remotely located box will receive the event later, by which time the puck could have entered the right hand partition, causing the virtual worlds to evolve inconsistently. A small inconsistency can multiply over time (especially if many inter-object interactions occur) to produce radically different versions of the virtual world at remote hosts[1].

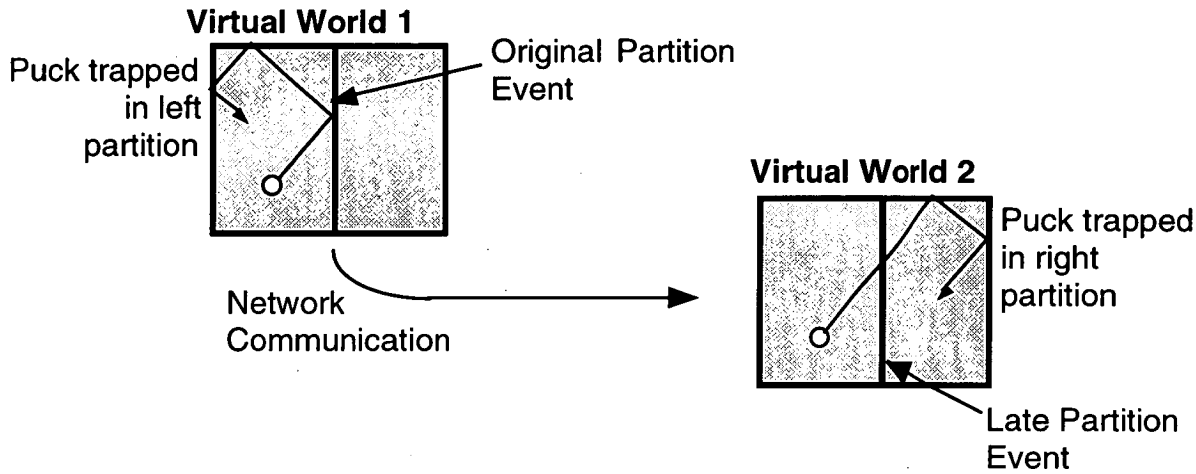


Figure 37 Inserting a Non-Deterministic Event into a Deterministic Simulation: Time Matters

To ensure 100% consistency, the simulation must rollback in time. The late event must then be inserted, and a roll forward in time needs to be performed to ensure that the user perceives time to be steadily increasing. A system that can insert events in this consistent manner will be able to eliminate network traffic caused by time dependent deterministic objects and will only have to send information about non-deterministic events. This is what the rollback/rollforward mechanism of the RTW system accomplishes very efficiently. The efficiency of the rolling process is due to the adaptation of the Time Warp mechanism for this purpose. Since the RTW mechanism is based on this Time Warp mechanism, it is necessary to know how the normal Time Warp mechanism works in order to understand the RTW mechanism:

4.2 Time Warp

The Time Warp mechanism (Steinman 1995, Jefferson 1985) is a parallel discrete event simulation mechanism. Discrete event simulations are composed of objects that send events to one another (or back to themselves). When an object processes an event, its state may change and one or more events may be scheduled, targeted at specific

[1] This is a well-known effect from Chaos theory, known as the 'butterfly effect'.

objects, at some time in the future. The processing of these events can cause other events to be scheduled, thus causing the simulation to evolve. Discrete event simulation mechanisms, such as the Time Warp, are suited to the construction (and parallel execution) of virtual worlds.

The Time Warp mechanism allows sets of objects to be simulated on various nodes (or processors) in a distributed computing environment. Objects communicate with each other by sending timestamped events indicating their scheduled execution time. Each node processes its events in timestamp order and keeps track of its own *local virtual time*. The local virtual time is the timestamp of the last processed event and is a representation of the progress the node has made through the simulation. Nodes process events *optimistically* and independently of the other nodes, thereby avoiding blocking delays and the associated synchronization messages caused by pessimistic methods.

Because there is no synchronization, when an interaction occurs between nodes the event sent by a node might arrive in the receiving node's past. This can be checked by comparing the receiving node's local virtual time and the incoming event's timestamp. The event could be late due to the receiving node processing events slower than the sending node or due to the network end-to-end delay experienced when sending the event across the network. One or more objects at the receiving node may have to be rolled back in time in order to process the late event in the required timestamp order. The Time Warp enables this rollback to occur efficiently.

The optimistic Time Warp mechanism performs rollbacks instead of maintaining strict synchronization as pessimistic distributed mechanisms do. The performance benefits of Time Warp simulation versus pessimistic simulation methods are analysed in a number of papers (Nicol & Fujimoto 1994). Time Warp simulation works by optimistically guessing that most of the time rollback will not occur, so the node can process events normally. If rollback does occur, due to a late event, a performance penalty is encountered. If rollback is a relatively infrequent, due to correct 'guessing', and the rollbacks are relatively small, Time Warp will perform better than pessimistic distributed simulation methods.

Figure 38 illustrates the communication between nodes and the rollback process. The event scheduled from node 0 arrives in the past of node 1 (that is, before its current local virtual time). The events processed after the scheduled time of this late event may need to be rolled back and recalculated because they may have different effects once the late event has been inserted. Note that the events scheduled after the local virtual time do not need to be rolled back because they have not been processed yet. Matters are complicated further by the fact that the events that are rolled back may have also scheduled further 'false' events (which in turn may have scheduled further false events) that may also need to be 'unscheduled'.

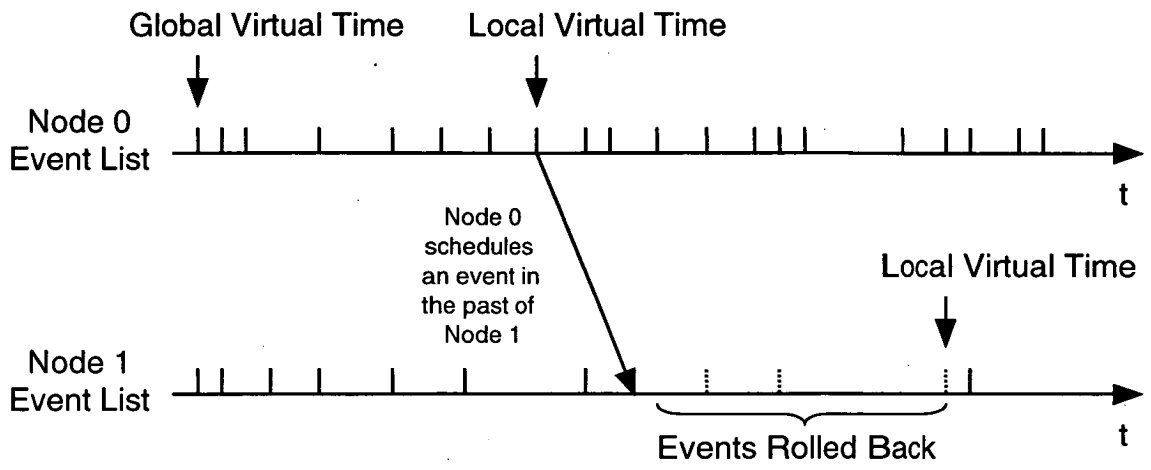


Figure 38 Optimistic event processing in Time Warp

Rollback

In order to support rollbacks in time when late events are received, the following information is stored for each Time Warp object (see Figure 39):

- *Object ID* — A unique object identifier.
- *Local Virtual Time* — The timestamp of the last processed event. This can be implemented as an index in the input event queue. It is also a representation of the progress the object has made through the simulation.
- *State* — The current state of the object.
- *State Queue* — The state of the object at various points in the past. Usually one is stored every time an event is processed, although this is not always required (see Preiss et al 1994). Techniques are available for reducing the memory requirements for state storage in Time Warp simulations and can be applied to the RTW simulations. One such technique is incremental state storage where the changes in state are stored, rather than the states themselves (Steinman 1995).
- *Input Event Queue* — A queue consisting of incoming events ordered according to their scheduled execution times. Old events are retained in this queue for rollback purposes.
- *Output Event Queue* — A queue of events generated by the object after processing events from the Input Event Queue. These are used during the rollback process to cancel secondary affects of late events.

Normal Event Processing

When an event is received that is not late, it is simply placed at the appropriate place on the input event queue. When the event becomes current and is about to be executed, the current state of the object is placed on the state store (see Figure 39).

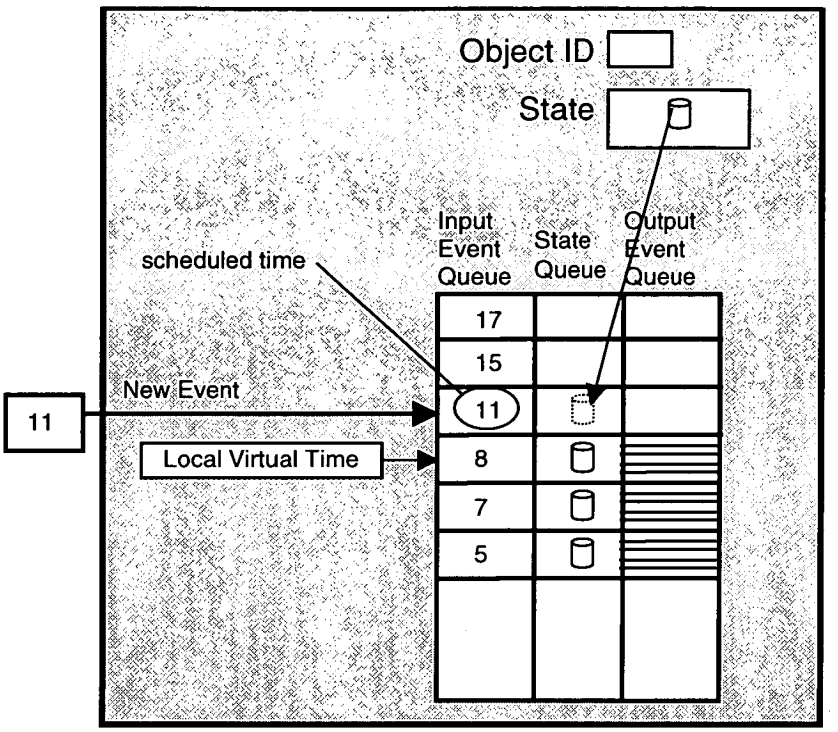


Figure 39 Before Executing the next Time Warp Event

During execution of the event the state of the object will most likely change, and becomes the new object state. Any events generated from the processing of the input event are sent to their appropriate objects and recorded on the output event queue (see Figure 40).

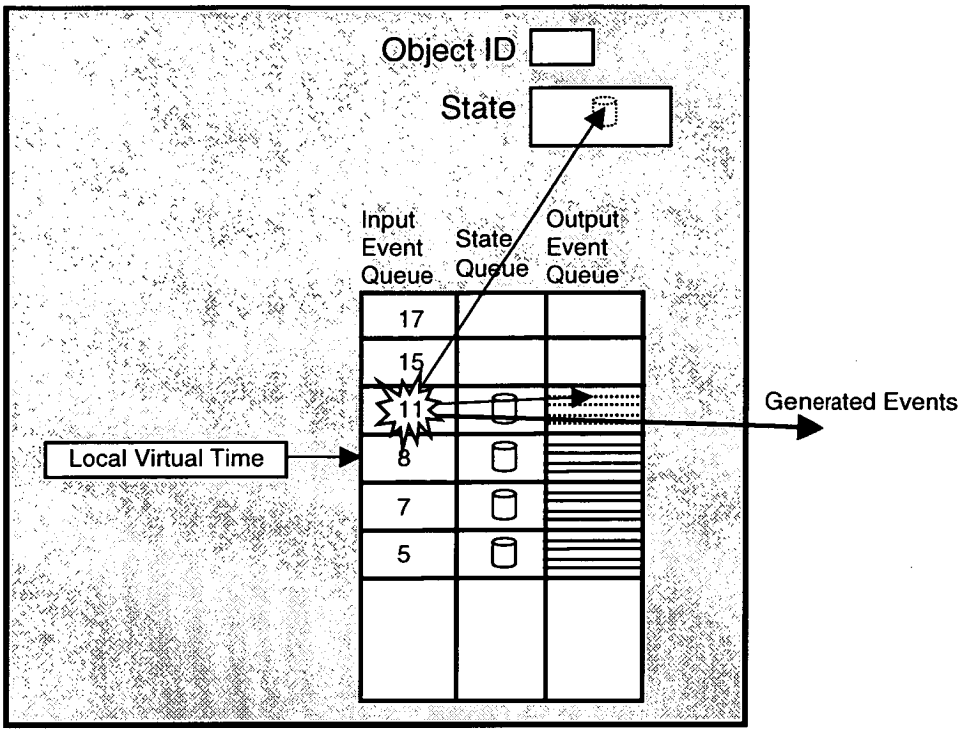


Figure 40 During Execution of the next Time Warp Event

Once the event has been executed, the object's local virtual time is set to the scheduled time of the just executed event (see Figure 41).

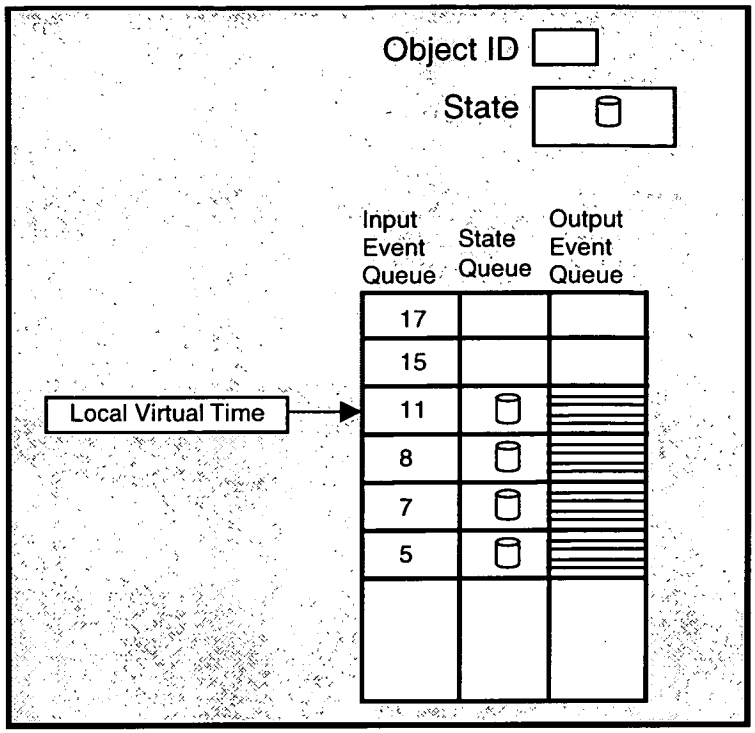


Figure 41 After Execution of the next Time Warp Event

Late Events

If an event arrives late (see Figure 42) the object will rollback in time so that the late event can be inserted correctly.

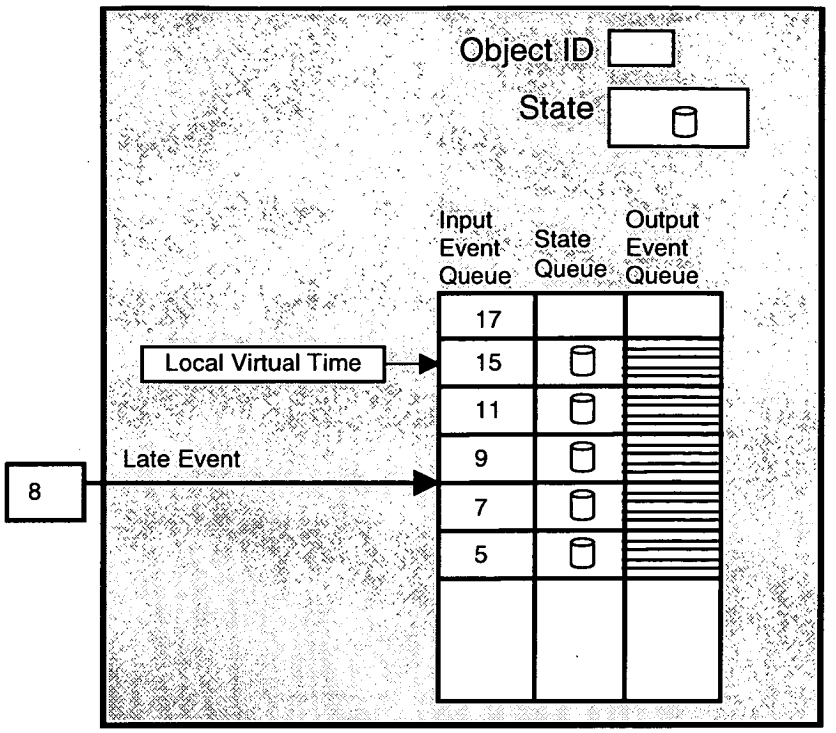


Figure 42 Before Rollback

A number of steps are involved in the rollback process (see Figure 43). During rollback:

- The state of the object is restored to the state just before the late event's scheduled time.
- The object's local virtual time is set to the time just before the late event's scheduled time.
- The late event is placed in the input event queue.
- Any events that have been scheduled by the object in the period rolled back (between the object's new local virtual time and the object's old local virtual time) could have been scheduled incorrectly because of the absence of the late event's effects. These can be cancelled by immediately sending out corresponding anti-events (called aggressive cancellation). Alternatively they can be cancelled by sending anti-events only when the output events are verified, during the rollforward, to have been sent incorrectly (called lazy cancellation, see Steinman 1995). Other objects are thus rolled back only as a secondary effect.

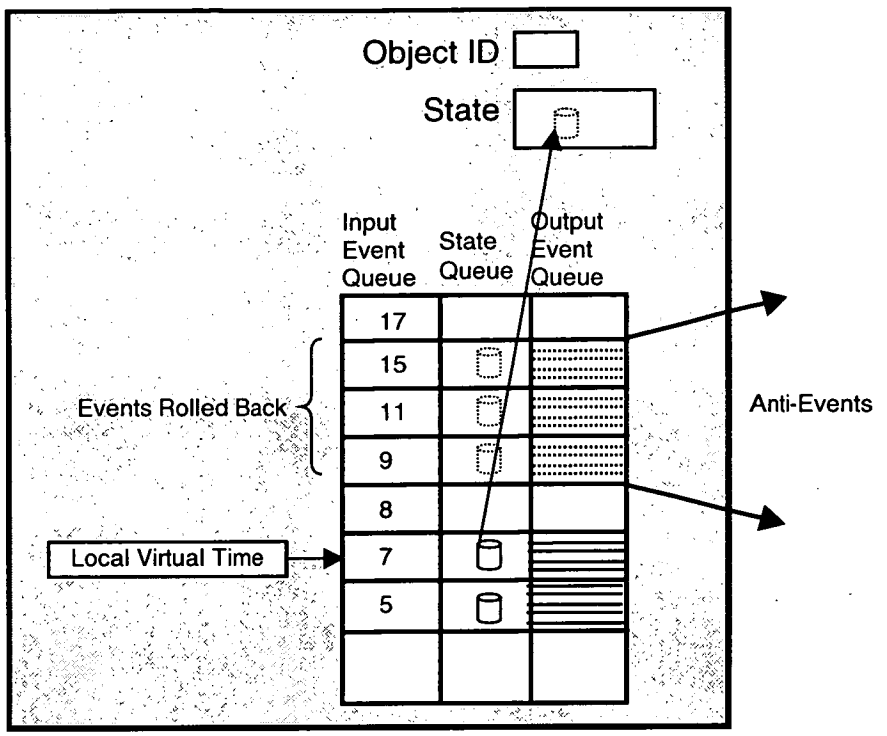


Figure 43 During Rollback

Once rollback has occurred, the simulation re-executes the (probably) incorrectly executed events that are still on the input event queue, in the normal Time Warp simulation manner (see Figure 44).

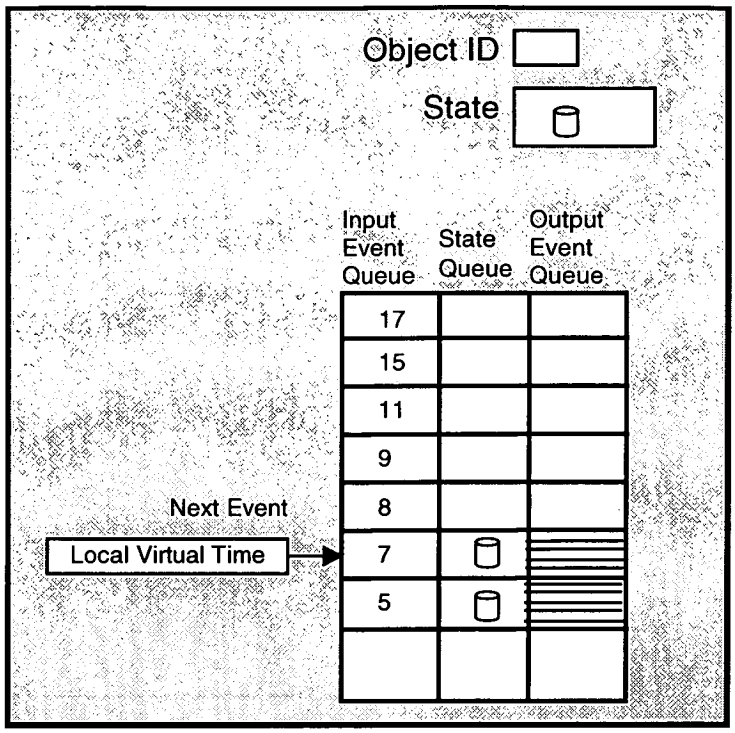


Figure 44 After Rollback

Anti-Event Rollback

The rollback procedure for early anti-events (see Figure 45) is slightly different to that of normal events: The anti-event (marked by the '-' sign) and its corresponding event annihilate each other (just as particles and anti-particles in physics annihilate each other). Figure 46 and Figure 47 show the same rollback procedure as for normal events except that the event scheduled at time '7' is annihilated by the anti-event.

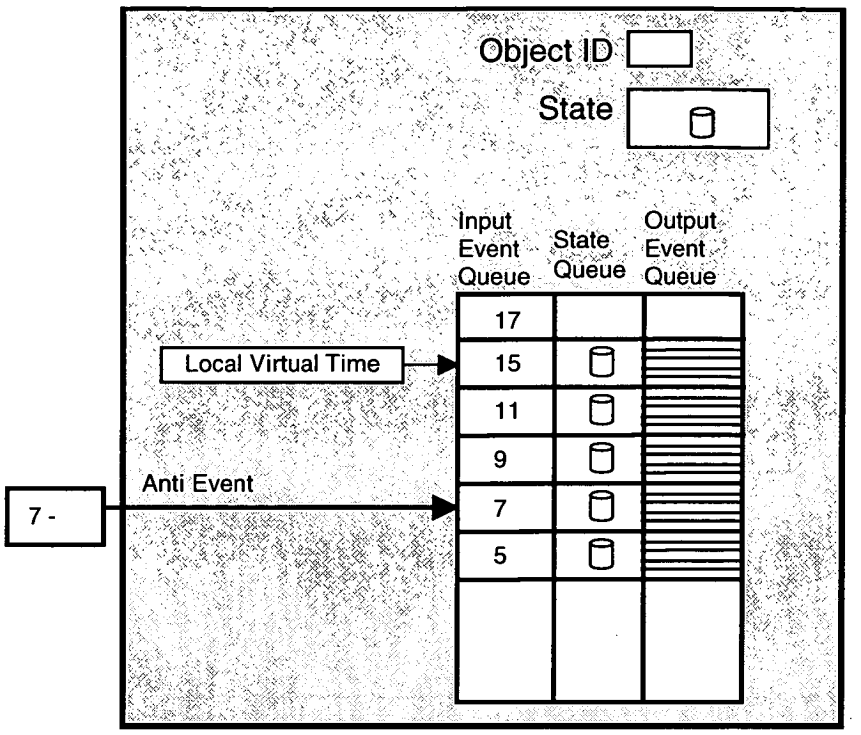


Figure 45 Before Rollback by an Anti-Event

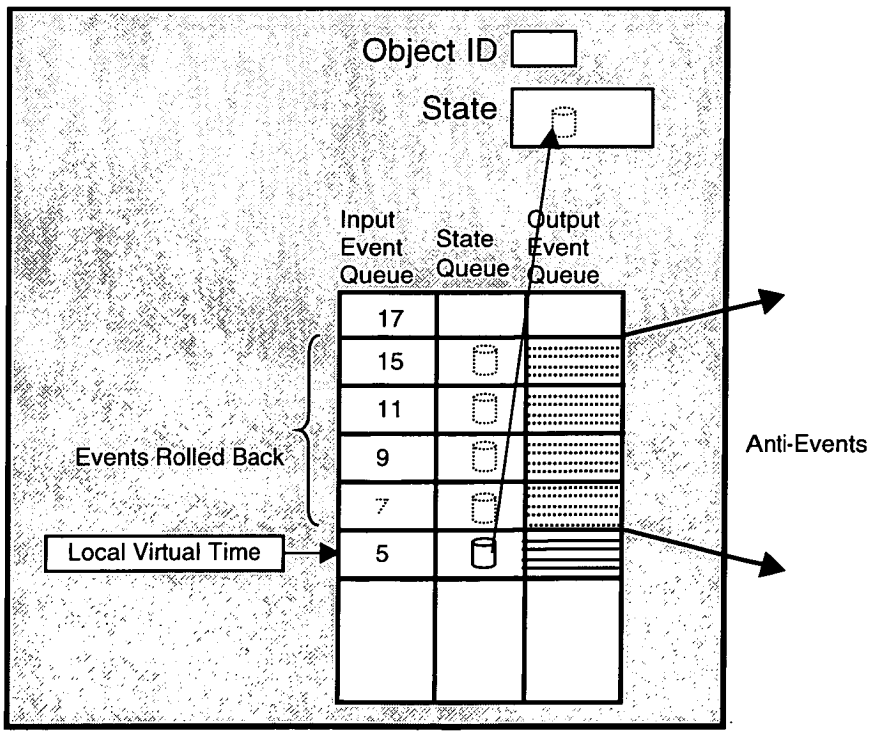


Figure 46 During Rollback by an Anti-Event

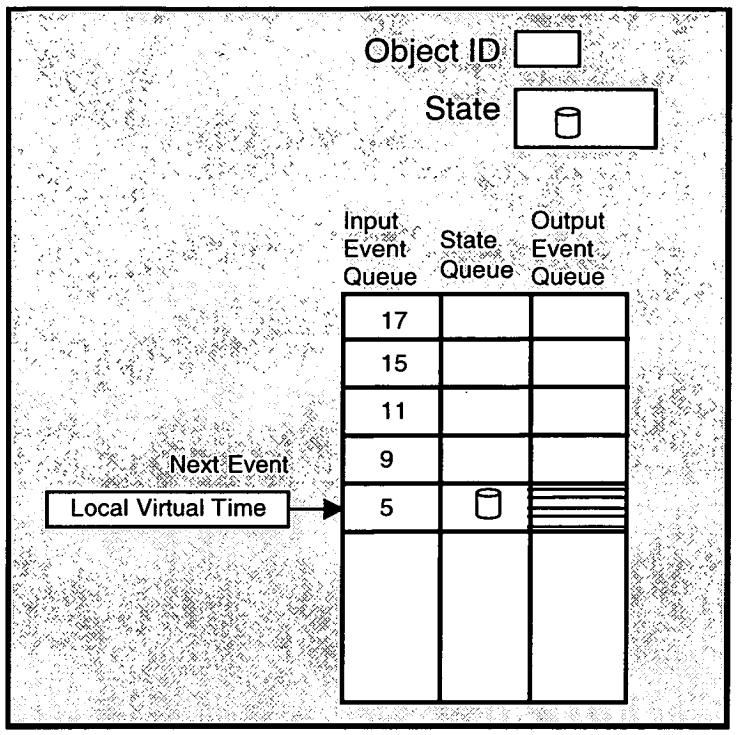


Figure 47 After Rollback by an Anti-Event

Early Anti-Events

In a parallel Time Warp simulation, it may happen that anti-events arrive before their corresponding events (see Figure 48). In this case, the anti-events are stored on the input event queue until the corresponding event arrives. Once the corresponding event arrives, it and the anti-event immediately annihilate each other (see Figure 49 & Figure 50). No rollback needs to occur.

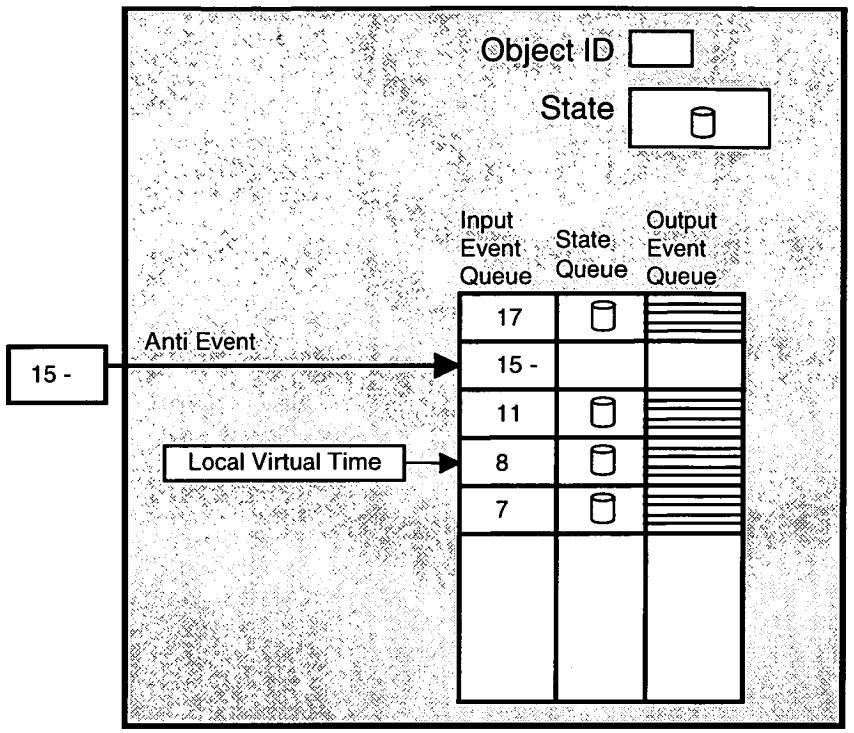


Figure 48 An Early Anti-Event

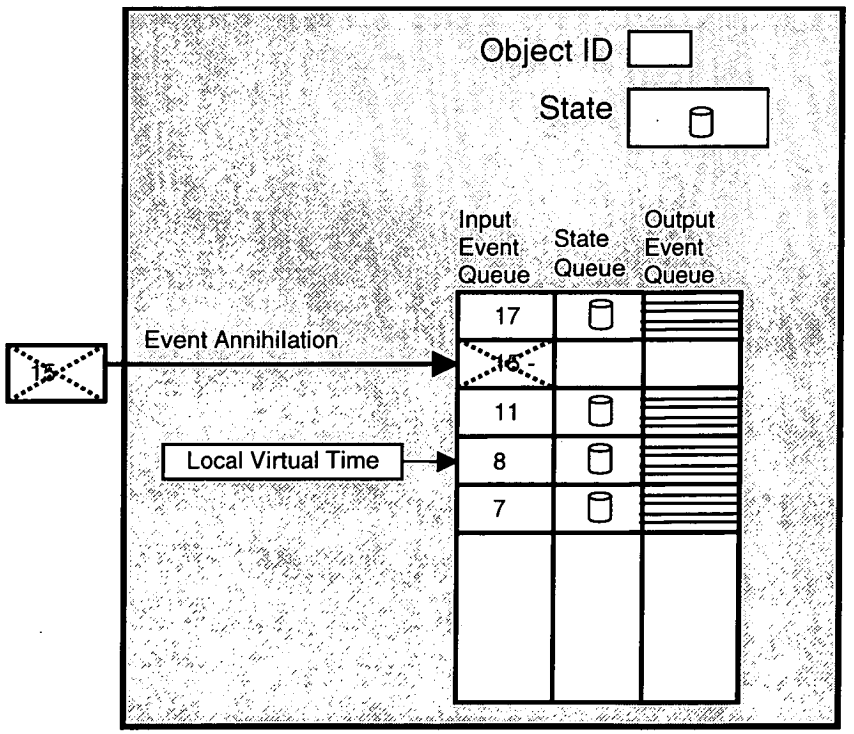


Figure 49 Annihilation with an Early Anti-Event

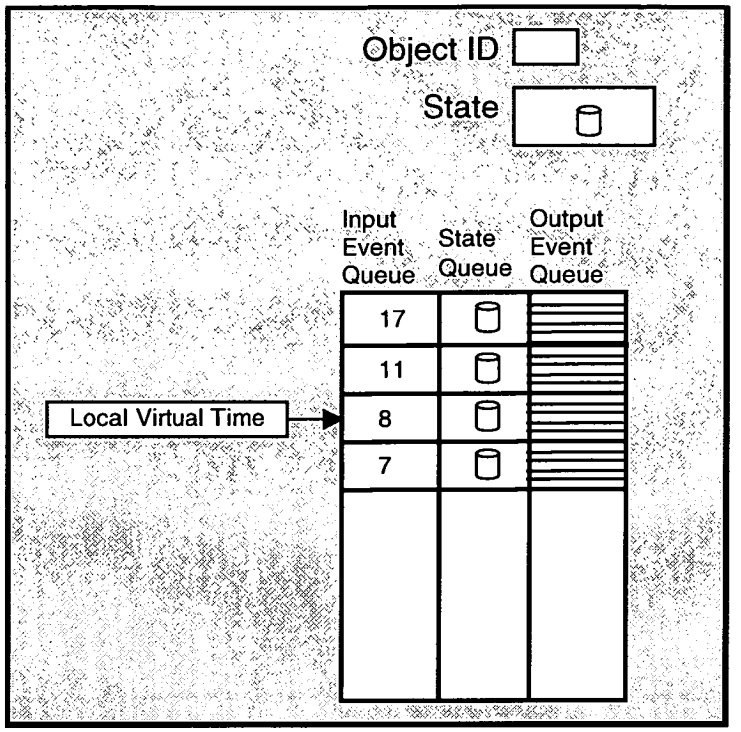


Figure 50 After Annihilation

Global Virtual Time

To enable rollback, the states of the Time Warp objects are required to be stored regularly. Since there is only a finite amount of memory available for storage, there needs to be some way of reclaiming memory from old states that are no longer needed once the simulation has been running for some time. In Time Warp simulation, Global Virtual Time (GVT) is calculated at regular intervals for this purpose. GVT is defined as the earliest scheduled time of an unprocessed event, or event in transit, in a distributed simulation (Steinman 1995). Rollbacks will thus not occur past GVT[1]. This means that any states earlier than GVT can be removed (garbage collected) to make room for future states (see Figure 51).

GVT is very expensive to calculate, in terms of communication costs, as it requires a message from every node stating their current local virtual time. It also needs to make sure that there are no events in transit over the network (usually by flushing all events from the network). Since most Time Warp simulations are run on local high speed networks available in parallel computers, the GVT communication costs are acceptable.

[1] Remember that events cannot be scheduled into the past.

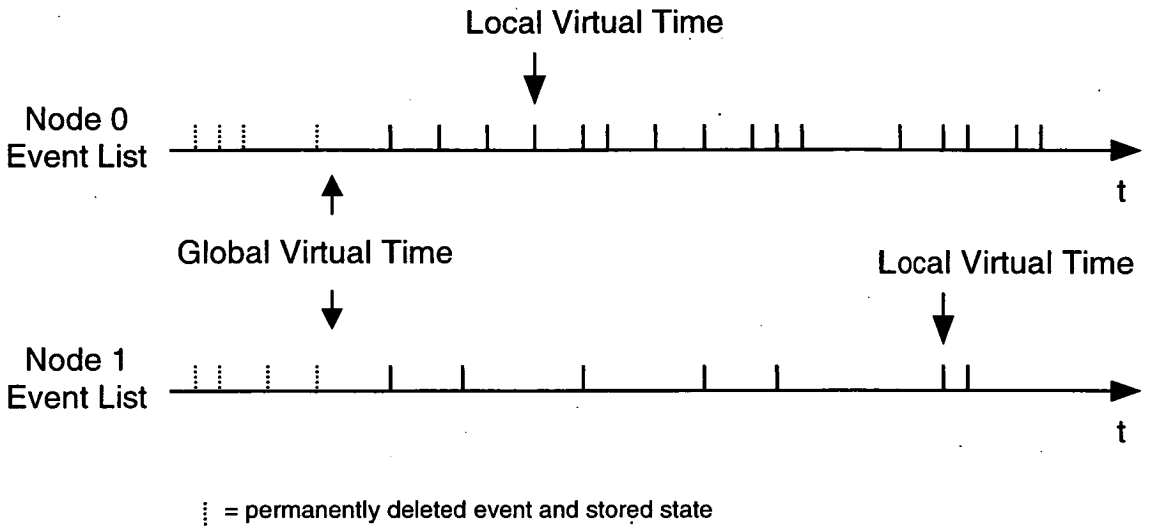


Figure 51 GVT and Garbage Collection

4.3 Replicated Time Warp

The Replicated Time Warp (RTW) architecture uses Time Warp as an efficient rollback mechanism for multi-user virtual reality systems. Rollback can be used for maintaining consistency between remote virtual worlds and reducing bandwidth consumption by eliminating synchronization messages for deterministic objects. The disadvantages of the RTW system are the extra processing and extra memory requirements for performing the rollbacks.

RTW is a more advanced implementation of Reversible Execution concurrency control. It satisfies the requirements of any Reversible Execution mechanism:

- Operations are executed immediately (see Clock Synchronization).
- Operations are globally ordered (see Rollback and Rollforward).
- When two or more operations have been executed concurrently, one or more of these operations may have to be undone and re-executed in the correct order (see User Interactions).

Clock Synchronization

In the Replicated Time Warp architecture, replicas of the entire Time Warp simulation are run on remote computers, typically one computer for each user. Each individual Time Warp simulation is synchronized with the computer's local real-time clock so that it operates in real-time. All remote computers' local real-time clocks are synchronized with each other. This clock synchronization provides timestamps for events to ensure global ordering of operations.

Inaccuracy in synchronization of the clocks can adversely affect the performance of the system by causing excessive rollbacks in the computers furthest in the future. Also, if a computer cannot process events fast enough to keep up with the local real-time clock then the simulation cannot be allowed to communicate with other simulations. This is because the events sent by the slow machine will cause excessive rollbacks on the machines that are keeping up with their local real-time clocks.

Several methods are available for synchronizing local real-time clocks over a network. The basic premise is to calculate the approximate network latency by sending a message and receiving an immediate reply. The latency is approximately half the round trip delay. The sent synchronization time can thus compensate for network latency. Systems such as the Network Time Protocol can get typical errors of less than 30 milliseconds over WANs (Coulouris et al 1994). Since non-deterministic objects are not likely to be updated much more than every 33 milliseconds (once per frame for a typical frame rate of 30 frames per second), an inaccuracy of 30 milliseconds means that clock inaccuracy will not account for much more than one rollback for each non-deterministic event inserted into the simulation: an acceptable statistic (see Replicated Time Warp, RTW Analysis for details on exactly how much this affects the simulation).

Rollback

RTW uses the same mechanism as the Time Warp system for rollback. Because other objects are rolled back only as a secondary effect and the rollback process is relatively simple, rollbacks can occur very quickly.

As with Reversible Execution concurrency control, dependencies between objects in the Time Warp should be kept to a minimum in order to avoid too many objects being rolled back by late events. This can be accomplished in Time Warp systems by making the granularity of objects small: each object is divided into smaller and smaller independent components. The objects should not be made too small, however, because there are small overheads for maintaining each separate object. In some cases it does not make sense to subdivide objects any further because operations are never performed at such a fine level of detail in the actual simulation.

An example of this issue of dependencies between objects can be seen when modeling collision detection within a virtual world. Consider a flat 2-dimensional box with a number of pucks bouncing around on its surface. In order for the pucks to collide with one another, they need to check their own positions against the positions of all the other pucks to see if a collision occurred. In this case, each puck is dependent on each other puck. This means that if one puck is rolled back, all the pucks will have to be rolled back.

A more efficient method that has fewer dependencies can be contrived by dividing the box into a number of partitions—each partition having its own collision manager (see Steinman & Wieland 1994 for more details). When a puck is located within a particular partition, it sends its position to that particular collision manager. The collision manager then checks that object's position against all other puck positions within that partition and informs them if they do collide. Using this method, pucks are only

dependent upon pucks within their own partition so when rollbacks occur only the objects within the particular partition are rolled back.

It is also possible for the RTW mechanism to handle periods of peak load better than other simulation mechanisms for multi-user virtual reality. This is due to the fact that it can optimistically simulate the virtual world ahead of time during periods when the processing power is available. The complex, time consuming portions of the simulation that could not be executed normally using conservative simulation may be able to be kept up with real-time because the optimistic RTW mechanism has already performed most of the calculations. Lazy cancellation can aid the process further by reducing the number of rollbacks (Lin & Lazowska 1991, Steinman 1995).

Rollforward

In order to avoid confusing the user when a rollback occurs, time should continue to appear to remain in sync with the local real-time clock. To do this, a rollforward of any rolled back objects to the current local real-time needs to occur as soon as possible. This involves the object executing its input events until its local virtual time has caught up with the local real-time clock.

Again, it is important that the dependencies between objects are kept to a minimum so that not too many objects need to be rolled forward. Lazy cancellation can also assist with the rollforward process by ensuring that objects only recalculate the effects of *changed* events.

The effects of the insertion of the late event and the rolling through time will cause anomalies visible to the user if the network latency is significant. The severity of the anomalies produced will be proportional to the network latency. This is the price paid for performance benefits of an optimistic approach and cannot be avoided (pessimistic approaches do not cause anomalies, but make you wait longer before they are performed—see the Concurrency Control section).

User Interactions

Traditionally in Time Warp simulation it is accepted that interactive input (external or non-deterministic events in the RTW system) cannot be rolled back. They are scheduled to be performed no later than GVT so that they are guaranteed not to be rolled back (Steinman 1995). This cripples the interactive performance of the simulation when network latency is large because GVT takes so long to compute.

Because the RTW system is intended to be used even when network latencies are large, these events are simply scheduled to occur immediately and will take part in the rollback process. User events are not tagged as being generated by any particular object. This means that there are never any corresponding anti-events for user interactions; once generated, they are permanently part of the simulation. The consequence of this method is that the user may act on an inconsistent virtual world state that exists before some late event is inserted. This may cause a problem because user events will be able to be produced in an illegal order, for instance an object could

be deleted and then moved. As with Reversible Execution, resolution must be performed to resolve conflicting operations (see Reversible Execution Concurrency Control).

User interactions must be transmitted reliably, otherwise simulations will evolve inconsistently. The method of communication is not important. For speed and efficiency, hardware multicasting will most often be the best choice, but unicasting can also be used where necessary.

Garbage Collection

In order to decide when to garbage collect old states in RTW, we need to isolate the cause of rollbacks. In RTW, the only external events that will cause rollbacks are non-deterministic interaction events. These would usually be sent using a reliable multicast protocol. The reliable multicast protocol needs to keep track of missing messages by placing sequence numbers on all messages. The receiving host checks for missing numbers in the sequence and then requests those messages to be resent.

Since the messages are sequenced, the receiving host knows that it has received all the messages from another host if it has received all messages, without any missing sequence numbers, up to a certain point. By scanning the scheduled time from the last message in an unbroken sequence from each host, the earliest timestamp of a non-deterministic event can be found. The RTW mechanism only needs to hold old states up to this timestamp. For example,

- Host 1 sequence numbers: 1,2,{3,scheduled time:211},_,5,6,7
- Host 2 sequence numbers: 1,2,3,{4, scheduled time:**172**}
- Host 3 sequence numbers: 1,2,3,4,5,{6, scheduled time:193}
- Host 4 sequence numbers: 1,2,3,{4, scheduled time:213},_,_,7,_,9

In the above example, the highest unbroken sequence number (missing sequence numbers are indicated by an '_') for each host is shown in brackets with its corresponding scheduled simulation time. The earliest scheduled simulation time is from Host 2. All stored states, input events and output events earlier than this time (scheduled time: 172) can be discarded safely, forever.

The overheads of deciding when to garbage collect can be avoided if large amounts of memory are available. States can simply be held onto for as long as possible—until the memory is used up. States, ideally, need only be stored for as long as the largest end-to-end delay because this is the maximum length of time it takes a user's interactions to be transmitted to the local host. This delay can occasionally be very long though, due to message loss on the network. If states are not stored for long enough, the system will not be able to roll back far enough and will no longer be able to guarantee consistency. It may well have to be removed from the simulation.

4.4 Related Work

Some pieces of work are closely related to the RTW mechanism. Each provides features needed for efficient multi-user virtual reality that the other does not possess.

Replicated Object in Time Warp Simulations provides:

- Deterministic objects that can change over time without providing synchronization information.

Reversible Execution provides:

- Synchronization of the simulations with local clocks for real-time operation.
- Having remote local real-time clocks synchronized with each other.
- Real-time interaction support, including resolution of conflicting operations.

When the features of each are combined, it is possible to create an efficient multi-user virtual reality system, such as the RTW.

Replicated Objects in Time Warp Simulations

Replicated Objects in a Time Warp system have been investigated before for the purpose of increasing the performance and fault tolerance of Time Warp simulations, and reducing message traffic (Agrawal & Agre 1992). It was not intended that the Time Warp simulations be fully replicated (although this would be possible). Because the simulations are not fully replicated, there is a need to send anti-events across the network. This partially replicated system provides valuable clues on how to implement partitioning (see Design Issues, Data Distribution) in the RTW system (see Conclusion, Future Work).

It was recognized in the study that Replicated Objects in Time Warp was an efficient way to maintain consistency between replicated Time Warp objects, especially when communication delays were significant and the ratio of read accesses over write access was high. This is certainly the case for multi-user virtual reality systems (read access is required around 30 times a second for rendering of the virtual world).

This system lacked the real-time facilities that are necessary for multi-user virtual reality interactivity:

- Synchronization of the simulations with local clocks for real-time operation.
- Having remote local real-time clocks synchronized with each other.
- Real-time interaction support, including resolution of conflicting operations.

These features are, however, present in Reversible Execution mechanisms.

Reversible Execution

Reversible Execution (see Design Issues, Concurrency Control for more details) is well suited to less dynamic applications than multi-user virtual reality systems, such as collaborative electronic whiteboarding. The reason for this is that previous implementations of reversible execution concurrency control algorithms have only been applied to objects that do not change over time (Floyd et al 1995, Karsenty & Beaudouin-Lafon 1993, Edwards & Mynatt 1997).

One of the reasons why they have not handled objects that do not change over time is probably because they have not been required to by the application. Distributed whiteboard applications usually involve only static drawings for instance. The whiteboard rollback mechanisms just order events according to timestamps. Rollback is performed by re-executing the entire event pipeline so that the late drawing operations are performed in the correct order. Each drawing operation just needs to be executed in timestamp order in order to maintain consistency among whiteboard replicas.

Due to the dynamic nature of virtual worlds, there are often a large number of objects that are required to be updated frequently (usually once per screen update: 10 to 60 times per second). Using these previous implementations would mean that these dynamic objects would have to send these updates many times per second across the network to interested peers. It can be seen that this severely limits the scalability of the system because the network traffic would quickly become a bottleneck as the number of dynamic objects increased. In order to support objects that can change over time, a more complex mechanism must be used that can insert events at the same simulation time at each multi-user virtual world, and rollback events and secondary effects that these may have caused over time.

4.5 RTW and Time Warping

Time Warping, as discussed in Chapter 2, not only smoothes the perceived effects of network end-to-end delay, but also reduces the *size* of rollbacks in the RTW mechanism.

There will be no rollbacks caused by non-deterministic events in a RTW system within an ideal Time Warping system because interaction with deterministic objects will, by definition, always be perfectly synchronized with the non-deterministic events. In practice, there will be variations from this perfect synchronization caused by network delay variation, lost network messages and network delay estimation errors. These will cause a small rollback if either the non-deterministic event arrives late, or the event will simply be queued and executed in the near future if it arrives early.

Time Warping reduces the size of rollbacks caused by non-deterministic events, but introduces a new cause of rollback—those caused by local deterministic objects.

Rollbacks may occur between deterministic objects, due to time synchronization differences created by the application of Time Warping. Deterministic objects near remote avatars will be simulated at a slightly delayed time compared to those in close proximity to the local avatar (see Figure 14). When objects near the local avatar interact with those nearer remote avatars, a rollback may occur. Because objects tend to interact with others in their close vicinity, the difference in synchronization times will usually be quite small. It will depend on the end-to-end delay of the nearby remote hosts and the equation used for calculating the Time Warping delay.

Because Time Warping reduces the size of rollbacks in the RTW, visibility of inconsistent states caused by rollbacks will be significantly reduced.

4.6 RTW Analysis

The main idea of RTW is that it trades *reduced bandwidth* for *increased local computation.* This section analyses this trade-off to see at what point the RTW becomes useful for given local processing hardware and given network connections.

Two different phases in computation will be looked at:

- Overheads during normal simulation for supporting rolling (state saving and recording dependencies between objects).
- Overheads of the rolling process itself.

General formulae will be developed for analysing any RTW system (see General Analysis). These will then be applied to a typical virtual world to obtain numerical results (see Specific Analysis).

General Analysis

General analysis of the performance of Time Warp systems is complex (Lin & Lazowska 1991, Nicol 1991, Lubachevsky & Weiss 1991, Dickens et al 1996) and not necessarily applicable to multi-user virtual reality RTW simulations. This is because they usually assume scheduled events are distributed randomly (see Figure 52).

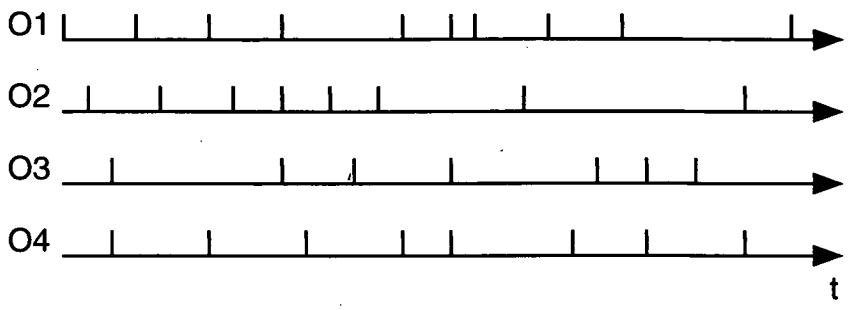


Figure 52 Events in Time Warp Systems are Usually Randomly Distributed

Due to the real-time nature of multi-user virtual reality simulations, most events will be distributed at regular intervals as required for smooth real-time animation (Figure 53) and collision detection (Figure 54)—the two most common operations in virtual reality systems.

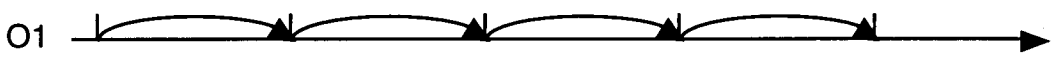


Figure 53 Virtual Reality Objects Animate at Regular Intervals

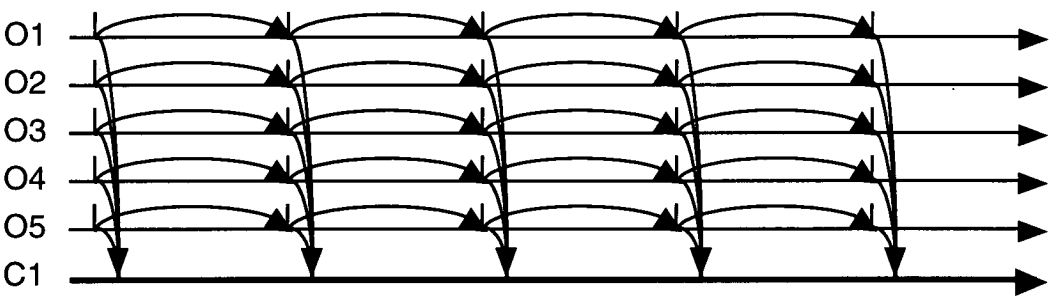


Figure 54 Virtual Reality Objects Provide Collision Information at Regular Intervals

Time Warp simulations are usually described as randomly distributed because updates for animation purposes are not generally required. This leaves the other events such as collision detection, which are likely to be distributed in a random distribution in virtual reality systems as well (see Figure 55).

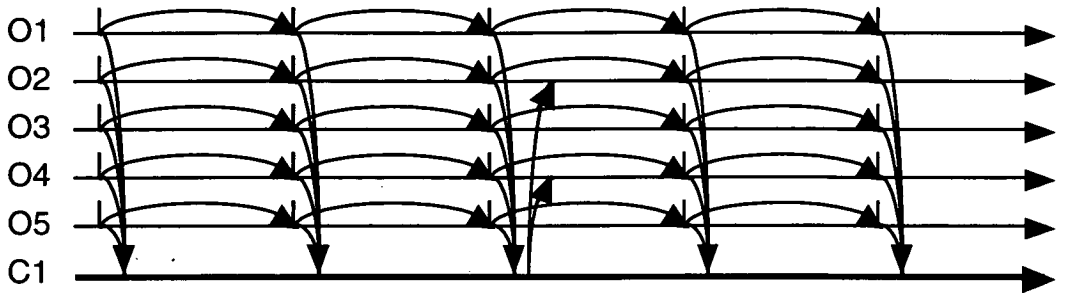


Figure 55 Virtual Reality Objects Collide at Irregular Intervals (O2 and O4 collided)

For simplicity of analysis the following will be assumed:

- A single processor architecture for the local machine—this will be the typical configuration of the systems the RTW is targeted at.
- A simple state storage mechanism whereby states are stored once each time an event is processed for a particular object. No compression techniques or other advanced state saving techniques (see Steinman 1995) will be taken into consideration.

Overheads for Supporting Rolling

Extra computational overheads for supporting rolling in the RTW are for:

- *State storage* — This computational time depends on the memory copy speed of the local machine and the size of the object's state. The memory size required is the size required to store the state of the objects.
- *Input event storage* — This computational time depends on the memory copy speed of the local machine and the size of the input event. The memory size required is the size required to store the event.
- *Output event storage* — Output event storage is minimal in a single processor (or shared memory) implementation because a pointer to the corresponding events in their input queues at their scheduled objects are kept.

The total extra computational overheads per simulated object for implementing RTW are:

$\tau_o = \tau_{ss} + \tau_{es} + \tau_{os}$

$\approx \tau_{ss} + \tau_{es}$, since $\tau_{ss} + \tau_{es} >> \tau_{os}$

where

τ_o = extra overheads for RTW

τ_{ss} = time for state storage

τ_{es} = time for input event storage

τ_{os} = time for output event storage

The total extra memory overheads per simulated object for implementing RTW:

$m_o = m_{ss} + m_{es} + m_{os}$

$\approx m_{ss} + m_{es}$, since $m_{ss} + m_{es} >> m_{os}$

where

m_o = memory overheads for the RTW

m_{ss} = memory for state storage

m_{es} = memory for input event storage

m_{os} = memory for output event storage

Overheads of the Rolling Process

The overheads of the rolling process consist of two phases:

- *Rollback* — The rollback involves restoring previous object states and deleting stored states and events that are no longer needed. Memory copies are fast and can be used for the state restoration, and memory deallocation can be made very fast (just pointer manipulation) if it is optimised.
- *Rollforward* — Rollforward involves recalculating the simulation of rolled back objects, plus overheads for state storage and event storage. The speed of the simulation recalculation depends on the speed at which individual objects can be simulated. This is entirely application, or more precisely, object dependent.

Inconsistencies in the virtual world can be visible due to network delay from remote hosts that are delivering their updates. When the updates arrive, a rollback and rollforward must occur before the inconsistencies are resolved. The period that the

virtual world will be inconsistent for is equal to the end-to-end delay plus the rolling time. The rolling time should thus be kept to a minimum.

The total extra computational overheads for rollback per object are:

$\tau_{rb} = (\tau_{rs} + (\tau_{ds}.n_{se})) + (\tau_{re} + (\tau_{de}.n_{se}))$

$\approx (\tau_{ds}.n_{se}) + (\tau_{de}.n_{se})$, since '$\tau_{rs}$' and '$\tau_{re}$' are simple pointer manipulation

$= (\tau_{ds} + \tau_{de}).n_{se}$

where

τ_{rb} = time to rollback an object

τ_{rs} = time to restore a state

τ_{ds} = time to delete a state

n_{se} = number of states and events rolled back

τ_{re} = time to restore an event

τ_{ds} = time to delete an event

The total extra computational overheads for rollforward per object are:

$\tau_{rf} = (\tau_s + \tau_o).n_{se}$

$= (\tau_s + (\tau_{ss} + \tau_{es})).n_{se}$, since $\tau_o \approx \tau_{ss} + \tau_{es}$

where

τ_{rf} = time to rollforward an object

τ_s = time to simulate an object

τ_o = overheads for supporting rolling

τ_{ss} = time for state storage

τ_{es} = time for event storage

n_{se} = number of states and events rolled back

The rolling process relies mainly on the speed of the rollforward because rollback can be optimised to be pointer manipulation:

$\tau_r = \tau_{rb} + \tau_{rf}$

$\approx \tau_{rf}$, since $\tau_{rf} >> \tau_{rb}$

$= \tau_s + (\tau_{ss} + \tau_{es})$

where

τ_r = rolling time for an object (rollback + rollforward)

τ_{rb} = time to rollback an object

τ_{rf} = time to rollforward an object

τ_s = time to simulate an object

τ_{ss} = time for state storage

τ_{es} = time for event storage

Memory overheads, on average, cancel out because the same number of states and events are reclaimed during the rollback as are used during the rollforward.

Specific Analysis

An example virtual world that simulates one hundred gas molecules bouncing around in a container will be analysed. It will give a good idea of the performance of the RTW in a multi-user virtual reality context. Molecules are assumed to be updated 10 times per second (the generally accepted minimum rate for smooth animation).

The numerical results quoted are those obtained from a Macintosh 7200/120 with a 120 MHz PowerPC 601 processor (see Appendix A: Profiling). This is close to 'typical' computer configurations quoted by Intel (1997), a 90 MHz Pentium.

Overheads for Supporting Rolling

The total extra computational overheads per simulated object for implementing RTW:

$\tau_o \approx \tau_{ss} + \tau_{es}$

where

τ_{ss} = 6.24e-6 seconds, assuming a 100 byte memory copy for event storage

τ_{es} = 3.12e-6 seconds, assuming a 200 byte memory copy for state storage

therefore

$\tau_o \approx 6.24e\text{-}6 + 3.12e\text{-}6$

= 9.36e-6 seconds

The extra memory overheads per simulated object:

$m_o \approx m_{ss} + m_{es}$

where

m_{ss} = 100 bytes for event storage

m_{es} = 200 bytes for state storage

therefore

$m_o \approx 100 + 200$

= 300 bytes

For the gas molecule simulation of 100 objects, updated 10 times per second,

- The time overhead is 9.36ms each second, or about 1% of the total processing time.
- The memory overhead is 300Kbytes per second of stored states and events, or about 1.5 Mbytes if events are stored for 5 seconds to allow for some dropped messages.

Overheads of the Rolling Process

Choosing a rollback of 500ms for this analysis will account for the end-to-end delay of one lost packet within Australia, or a typical rollback with an intercontinental connection (both with modem access). If the gas molecules are updated 10 times per second, this will mean the rollback of 5 events.

From the general analysis we know that:

$\tau_{rb} \approx (\tau_{ds} + \tau_{de}).n_{se}$

$= (2.93e\text{-}6 + 1.97e\text{-}6).5$

$= 24.5e\text{-}6$ seconds

From a prototype implementation, about 1000 molecules could be simulated with updates 10 times per second (including the time taken for scheduling new events). The simulation time per object was thus 0.1ms.

$\tau rf = (\tau s + \tau o).n_{se}$

$= (0.1e\text{-}3 + 9.36e\text{-}6).5$, 'τ_o' obtained from calculation above

$= (0.11e\text{-}3).5$

$= 0.55$ ms

Note that the simulation time is about 10 times larger than the time required to perform the RTW overheads of state and event storage.

Over the 500ms period the total time it takes to roll the puck forwards is 0.55ms. Therefore the total time taken in the rollback/rollforward process per object is:

$\tau_r = \tau_{rb} + \tau_{rf}$

$= 0.1 + 0.55$ ms

$= 0.65$ ms

So when a reliable multicast message arrives very late (500ms), in the best case only one object is rolled and:

$\tau_r = 0.65$ ms

In the worst case, when dependencies between objects is very high, all objects will be rolled back and:

$\tau_r = (0.65e\text{-}3).n_o$

$= (0.65e\text{-}3).100$

$= 65$ ms

where

n_o = total number of objects in the RTW simulation

The RTW therefore adds a 65ms worst-case delay to the 500ms network delay in return for eliminating network synchronization traffic from the gas molecules. In the best case it accounts for 0.65ms—a negligible amount of time.

The above results are given using the default memory management routines as supplied by the compiler and accessed via the C++ code. Performance could be improved greatly by using custom memory management routines. For event storage for instance, events could be constrained to a maximum size (ideally, all events would be the same length). Memory allocation and deallocation would then be a simple case of having a free-list of available event memory. Memory allocation would involve putting the event in the first available spot (obtained from the free-list) and removing it from the list. Memory deallocation would simply be adding that spot to the free list.

With state storage, it would be much more restricting having constraints on the sizes of states, since object states sizes will naturally vary a lot. It would still be possible to optimise the memory management further than the default memory management routines.

Discussion

From the above rough calculations, it can be seen that the overheads for supporting rollback within the gas molecule simulation with 100 objects are acceptable:

- About 1% of the total processing time.
- About 1.5 Mbytes if events are stored for 5 seconds to allow for some dropped messages.

The RTW adds a 65ms worst-case delay to a 500ms network delay in return for eliminating network synchronization traffic from the gas molecules. In the best, and most common case, it accounts for 0.65ms—a negligible amount of time.

Time Warping can also be used to further reduce the size of rollbacks. The size of the rollbacks will be given by the reliable multicast delay variation if Time Warping is used. A significant delay will be observed when multicast messages are lost and need to be retransmitted.

As mentioned before, the main idea of RTW is that it trades reduced bandwidth for increased local computation. We can now see at what point the RTW becomes useful, when compared to dead-reckoning, for given local processing hardware and given network connections.

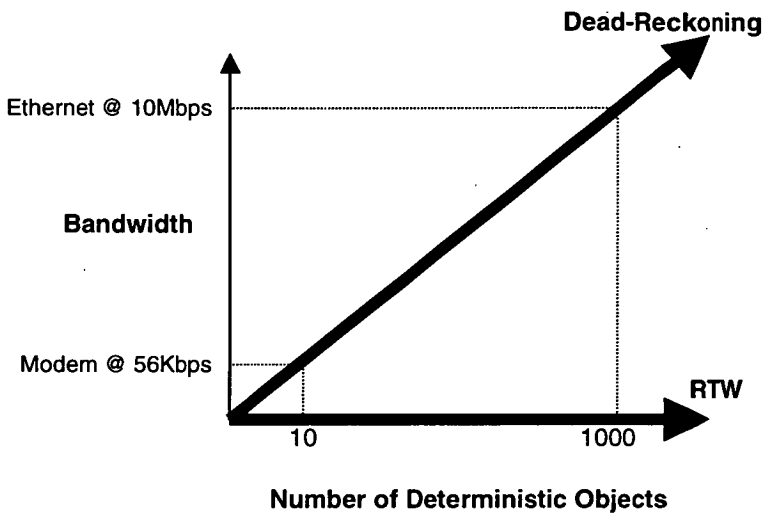


Figure 56 Bandwidth versus Number of Deterministic Objects

Figure 56 illustrates that RTW can support any number of objects, even over the lowest bandwidth connections. Dead-reckoning (used by most of the reviewed systems in this thesis) can support a maximum of about 10 objects over a modem connection, and an absolute maximum of about 1000 objects over an Ethernet connection. This figure is more likely to be just over 300 objects when using the standard DIS protocol (Zeswitz 1993, Macedonia et al 1994).

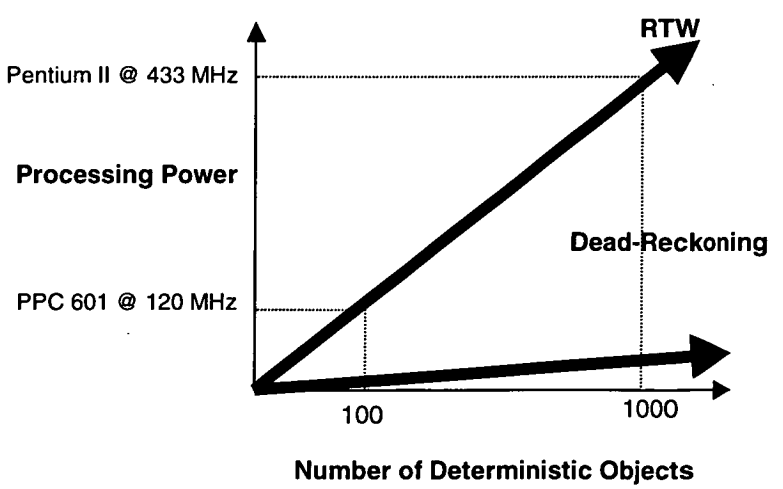


Figure 57 Processing Power versus Number of Deterministic Objects

Bandwidth is one of two limitations. Local host processing power is the other limitation. A system can only support the maximum number of objects that satisfy both limitations. From Figure 57 it can be seen that dead-reckoning is very unlikely to be limited by processing power. Even a relatively low powered computer will be able to compute thousands of dead-reckoned objects (Pratt 1993). RTW, on the other hand, is restricted by processing power, but not by bandwidth. The Specific Analysis section shows that 100 objects can easily be supported by a typical personal computer.

With multi-user virtual reality systems using dead-reckoning, the limit of how many objects can be simulated with some given processing power will be defined by factors other than the processing caused by the dead-reckoning algorithm. The largest factor is most likely to be the graphics processing overheads. For 3D graphics, this includes geometry transforms and texture mapping for the objects concerned. These are also likely to take up at least as much processing time as the RTW takes, and prototype implementations show this to be true.

Prototype Implementation

Prototypes of the RTW were implemented in AppleScript, C, C++ and Java.

The AppleScript version was a proof of concept prototype and helped establish the correct algorithm for handling user interaction within the RTW mechanism, and initial synchronization from a newly joined host. Development was fast due to the advanced features provided by this high level language (such as passing script objects between applications, including transparently over a network). This prototype was far too slow to test whether the rolling process could occur in real-time.

The Java implementation (see Figure 58) was the quicker to implement. This was because of the very good TCP/IP integration, relatively simple GUI code, and built in data structures. The major parts of the Java implementation consisted of:

- The ability to support multiple RTW simulations on the same computer (communicating though the network interface).
- A separate server thread to transmit initial world state (via the serialization interface) to new hosts without halting the current local simulation.
- Client code to synchronize with remote virtual world server threads.
- Clock synchronization code.
- GUI code for input and graphical output (2D).
- Time Warp implementation for rolling.
- Test virtual world with a bouncing puck.

Due to the large effort required to implement the RTW architecture there was not enough time left to implement a significant test virtual world. The disadvantage of the

Java implementation was the lack of speed and the inability to obtain profiling information on the speed of memory allocation and deallocation.

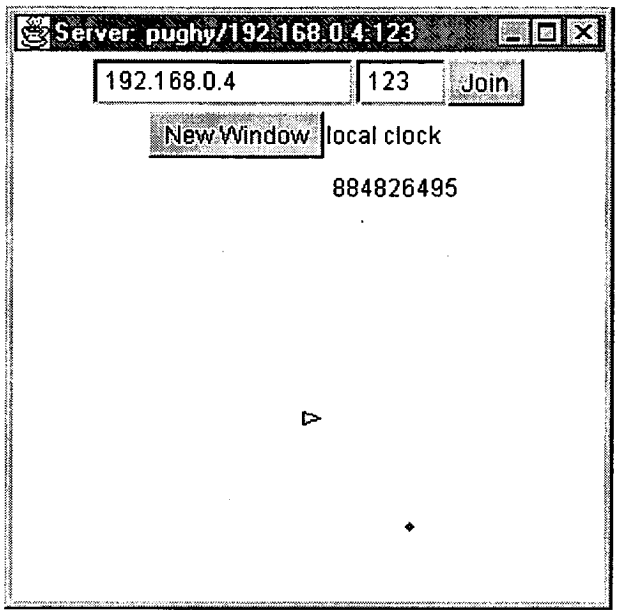


Figure 58 Java Prototype Implementation Screenshot

The C and C++ implementations progressed well and supplied the same functionality as the Java version, but became complicated when threading, networking and GUI code was added. Testing required the ability to create more than one window and have separate simulations run in each. Fake network code was implemented that emulated reliable multicasting and allowed the windows to communicate with each other. The emulated network allowed the end-to-end delay to be specified at runtime and would also count and display the number of message sent from the window.

The test virtual world consisted of pucks bouncing around on a two-dimensional surface. The pucks were updated ten times each second and could be interacted with by colliding into them with a user controlled, dead-reckoned 'space-ship'. In this version the pucks could bounce off each other, but this would result in all the pucks rolling back because an efficient collision detection algorithm was not used (see section 4.3 Replicated Time Warp, Rollback).

Qualitative results showed that with a simulated end-to-end delay of 300ms, rolling was barely noticeable, especially if only one puck was rolled back. Once more than ten pucks were rolled back, the delay became easily noticeable. Delays of three to five seconds were tested and the simulation could catch up in less than a second if a single puck was rolled back. Inconsistencies in the user interface were obvious with the large delay (no Time Warping was used). If a large number of pucks were simulated and all of them were rolled back, rolling could take a few seconds (two or three) to complete. The fake network code's message counter verified that deterministic simulations could

remain consistent by sending only non-deterministic interactions. These results are consistent with the RTW analysis presented in the previous section.

4.7 Summary

This chapter presented the background of the Replicated Time Warp system and explained how it works. It analysed and discussed its performance compared to dead-reckoning. Conclusions were drawn about its effectiveness compared to the other systems, in particular, how bandwidth restrictions are lifted from multi-user virtual reality systems taking advantage of Replicated Time Warp.

5 CONCLUSION

The aim of producing a scalable, high performance software architecture for modem and mobile communications users that solves concurrency and consistency issues has been achieved. The RTW maintains 100% consistency while lifting bandwidth restrictions that limit modem simulations using alternate synchronization methods to around ten dynamic objects. RTW also performs very well. Its response and notification times are optimal, and its commit times are acceptable for most applications.

The RTW system has maximum performance benefits when a large number of simple deterministic objects are simulated by fast computers connected by low bandwidth connections. RTW also performs optimally in terms of minimizing the effects of end-to-end delay. This makes it ideal for use on the Internet where bandwidth is scarce, but relatively powerful personal computers are connected that can handle the demands of the RTW system.

The system will allow developers to create far richer and more dynamic multi-user virtual worlds than could have been possible before on low bandwidth networks.

Summary

RTW Advantages:

- RTW eliminates network traffic caused by deterministic objects. Other systems, such as those described in the Related Work chapter, need to update deterministic objects periodically in order to maintain consistency between remote simulations.
- RTW's optimistic simulation can perform better than pessimistic simulation methods by computing results ahead of time and rolling back if wrong guesses are made. This feature is not provided by other systems.
- RTW minimizes network delays by offering optimal notification times. This is the same as other peer-to-peer concurrency control systems.
- RTW has optimal response times because operations are always reflected locally immediately. The systems described in the Related Work chapter are often required to wait for locks and cannot reflect operations immediately.
- RTW can be easily implemented in a parallel computing architecture due to its grounding on the Time Warp simulation mechanism.

RTW Disadvantages:

- RTW requires more processing power and has heavier memory requirements to be able to maintain the ability to perform rollbacks. Mechanisms such as dead-reckoning with peer-to-peer master entities require less memory and processing.
- RTW can display brief user-interface inconsistencies due to its non-zero commit times. Other systems make you wait before executing an operation, rather than allowing processing and then rolling back.

Future Work

Firstly, the RTW could be extended to support partitioning (as discussed in the Data Distribution section of the Design Issues). Because not all objects are replicated at each host when partitioning is used, anti-messages need to be sent across the network. An ideal system would figure out the optimal partitioning of objects based on the given local computational power and the dependencies between objects:

- If lots of computational power is available, more objects can be simulated locally and less network communication is required.
- If certain objects have tightly coupled interactions, they should be simulated on the same host to avoid excess network communication.

Secondly, a locking concurrency control scheme could be merged with the RTW to provide more flexibility for virtual world designers and better performance for the end users. Locking provides better commit times than plain RTW once a lock has been granted, but restricts other users' access to the locked objects. It would be nice if the designer could choose which scheme to use for different objects.

Finally, a full-blown implementation of the RTW with a significant test virtual world would provide many insights into the practical implications of the system. Theoretical analysis can easily overlook deficiencies in a complex system such as the RTW.

6 APPENDIX A: PROFILING

Profiling Test Programs

A small suite of programs was developed specifically to test the speed of the computer to perform operations specific to the RTW algorithm. These were developed as an indication of what results would be possible with optimised code. They were used because they required less effort to develop than would be required to optimise the code already developed in other RTW prototypes.

The tests performed were memory allocation and deallocation tests. They were implemented on a Power Macintosh 7200/120 (120Mhz PowerPC 601) in C++ using Metrowerks CodeWarrior Release 10. Large blocks of memory were used for testing to eliminate any speedup from the level 1 or level 2 caches in the computer.

Profiling Results

τ_{es}	= time for event storage
	= 100 bytes * 31.2e-9 per bytes
	= 3.12e-6 seconds (= max 320 000 per second)
τ_{ss}	= time for state storage
	= 200 bytes * 31.2e-9 per bytes
	= 6.24e-6 seconds (= max 160 000 per second)
τ_{ds}	= time for delete state
	= 2.93e-6 seconds
τ_{de}	= time for delete event
	= 1.97e-6 seconds

7 REFERENCES

Agrawal, D. & Agre, J. R. 1992, 'Recovering from Multiple Process Failures in the Time Warp Mechanism', *IEEE Transactions on Computers*, Vol. 41, No. 12, December 1992.

Agrawal, D. & Agre, J. R. 1992, 'Replicated Objects in Time Warp Simulations', *Proceedings of the 1992 Winter Simulation Conference.*

Anderson, D., Barrus, J., Howard, J., Rich, C., Shen, C. & Waters, R. 1995, 'Building Multi-User Interactive Multimedia Environments at MERL', *IEEE MultiMedia*, Winter, 1995.

Aronson, J. 1997, 'Dead Reckoning: Latency Hiding for Networked Games', *Game Developer: Online Gaming*, Winter 1996/1997, Miller Freeman Inc, ISSN 1073-922X. http://www.gdmag.com/

Bailey, R. W. 1982, *Human Performance Engineering: A Guide for System Designers.* Englewood Cliffs, NJ: Prentice-Hall.

Bangun, R. A. & Beadle, H. W. P. 1997, 'Traffic on a Client-Server Based Architecture for Multi-User Network Game Applications', *Proceedings IEEE/IEE International Conference on Telecommunications*, Melbourne, April 1997, to appear.

Barrus, J. W., Waters, R. C. & Anderson, D. B. 1995, 'Locales and Beacons: Efficient and Precise Support For Large Multi-User Virtual Environments', Technical Report, Mitsubishi Electric Research Laboratories, Cambridge Research Center, Cambridge, Massachusetts, http://www.merl.com/TR/TR95-16/Welcome.html

Carlsson, C. & Hagsand, O. 1993, 'DIVE — A Multi User Virtual Reality System', *IEEE Annual Virtual Reality International Symposium*, September 1993, IEEE Service Center, ISBN 0-7803-1364-X.

Cheshire, S. 1996, 'Latency and the Quest for Interactivity', White Paper commissioned by Volpe Welty Asset Management, L.L.C., mailto:cheshire@cs.stanford.edu, http://ResComp.Stanford.EDU/~cheshire/

Chowdury, S., Bluestein, W. M. & Davis, K. S. 1997, 'Internet Games', The Forrester Report: Entertainment & Technology Strategies, Vol.1, No. 1, http://www.forrester.com/

Coulouris, G., Dollimore J. & Kindberg T. 1994, *Distributed Systems Concepts and Design*, Queen Mary and Westfield College, University of London, 2nd Edition, Addison-Wesley, ISBN 0-201-62433-8.

Das, T. K., Singh, G., Mitchell, A., Kumar, S. & McGee, K. 1997, 'NetEffect: A Network Architecture for Large-scale Multi-user Virtual Worlds', VRST'97, September 1997, http://www.historycity.org.sg/neteffect/papers.htm

Das, T. K., Singh, G., Mitchell, A., Kumar, S. & McGee, K. 1997, 'Developing Social Virtual Worlds using NetEffect', VRST'97, June 1997, http://www.historycity.org.sg/neteffect/papers.htm

Davis, P. K. 1995, 'Distributed Interactive Simulation in the Evolution of DoD Warfare Modeling and Simulation', *IEEE Computer Graphics and Applications*, Vol. 83, No. 8, August 1995.

Deering, S. 1989, 'Host Extensions for IP Multicasting', Network Working Group, Request for Comments: 1112.

Dickens, P. M., Nicol, D. M., Reynolds, P. F. & Duva, J. M. 1996, 'Analysis of Bounded Time Warp and Comparison with YAWNS', *ACM Transactions on Modeling and Computer Simulation*, Vol. 6, No. 4, October 1996, pp. 297-320.

Edwards, W. K. & Mynatt, E. D. 1997, 'Timewarp: Techniques for Autonomous Collaboration', CHI'97, http://www1.acm.org:82/sigs/sigchi/chi97/proceedings/paper/wke.htm

Ellis, C. A., Gibbs, S. J. & Rein, G. L. 1991, 'Groupware: Some Issues and Experiences', *Communications of the ACM*, Vol. 34, No. 1, January 1991.

Fall Virtual Reality World '95: The Complete Conference on CD-ROM, Fall 1995, Canyon Interactive.

Fitzsimmons, E. A. & Flether, J. D. 1995, 'Beyond DoD: Non-Defense Training and Education Applications of DIS', *IEEE Computer Graphics and Applications*, Vol. 83, No. 8, August 1995.

Floyd, S., Jacobson, V., McCanne, S., Liu, C. & Zhang, L. 1995, 'A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing', *ACM SIGCOMM '95*, pp. 342-356.

Fluckiger, F. 1995, *Understanding Networked Multimedia: Applications and Technology*, Prentice Hall, ISBN 0-13-190992-4.

Funkhouser, T. A. 1995, 'RING: A Client-Server System for Multi-User Virtual Environments', *Computer Graphics (1995 ISGGRAPH Symposium on Interactive 3D Graphics)*, Monterey, CA, April 1995, pp. 85-92.

Funkhouser, T. A. 1996, 'Network Topologies for Scalable Multi-User Virtual Environments', *IEEE VRAIS '96*, San Jose, CA, April 1996

Gautier, L. & Diot, C.1998, 'Design and Evaluation of MiMaze, a Multiplayer Game on the Internet', IEEE Multimedia Systems Conference, Austin, June 28 – July 1, 1998, http://www.inria.fr/rodeo/MiMaze/.

Gossweiler, R., Laferriere, R. J., Keller, M. L. & Pausch, R., 'An Introductory Tutorial for Developing Multi-User Virtual Environments', PRESENCE: Vol. 3, No. 4: Special issue on Networked Virtual Environments & Teleoperation, http://www.cs.virginia.edu/~rg3h/networkVR/paper.html.

Hagsand, O. 1996, 'Interactive Multiuser VEs in the DIVE System', *IEEE Multimedia*, Volume 3, Number 1, Spring 1996, pp. 30-39.

Heim, R. 1997, 'Terminal Trends', Siemens Telecom Report, http://www.siemens.de/

Hofer, R. C. & Loper, M. L. 1995, 'DIS Today', *IEEE Computer Graphics and Applications*, Vol. 83, No. 8, August 1995.

Holbrook, H., Singhal, S. K. & Cheriton, D. R. 1995, 'Log-Based Receiver-Reliable Multicast for Distributed Interactive Simulation', to be presented at SIGCOMM 95.

Honda, Y., Mitra, Rockwell, B. & Roehl, B. (eds) 1997, *Living Worlds: Making VRML 2.0 Applications Interpersonal and Interoperable*, http://www.livingworlds.com/

IEEE P1516.1, M&S HLA - Federate I/F Spec, DRAFT 1, HLA Training CD 1999

IEEE P1516/D1, Draft Standard [for] Modeling and Simulation (M&S), High Level Architecture (HLA) – Framework and Rules, HLA Training CD 1999

Intel, 1997, 'Multiplayer Internet Gaming', Developer Relations Group, Hybrid Application Cookbooks, http://developer.intel.com/drg/hybrid_author/cookbooks

Jefferson, D. R. 1985, 'Virtual Time', *ACM Trans. Programming Languages and Systems*, Vol. 7, No. 3, July 1985, pp. 404-425.

Karsenty, A & Beaudouin-Lafon, M. 1993, 'An Algorithm for distributed Groupware Applications', *Proceedings of the 13th International Conference on Distributed Computing Systems*, IEEE, pp. 195-202.

Koch, M. 1995, 'The Collaborative Multi-User Editor Project Iris', Institut Für Informatik, Technische Universität München.

Koifman, A. & Zabele, S. 1996, 'RAMP: A Reliable Adaptive Multicast Protocol', Fifteenth Annual Joint Conference of the IEEE Computer and Communication Societies, San Francisco, CA, March 26-28, 1996.

Lamport, L. 1978, 'Time, clocks and the ordering of events in a distributed system', *Communications of the ACM*, Vol. 21, No. 7, July 1978.

Leivent, J. I. & Watro, R. J. 1993, 'Mathematical Foundations for Time Warp Systems', *ACM Transactions on Programming Languages and Systems*, vol. 15, no. 5, November 1993, pp. 771-794.

Lin, Y. & Lazowska, E. D. 1991, 'A Study of Time Warp Rollback Mechanisms', *ACM Transactions on Modeling and Computer Simulation*, Vol. 1, No. 1, January 1991, pp. 51-72.

Locke, J. 1994, 'An Introduction to the Internet Networking Environment and SIMNET/DIS', unpublished work, Computer Science Department, Naval Postgraduate School, Monterey, California, http://www.npsnet.nps.navy.mil/npsnet0/publications.html

Lu, G. 1996, *Communication and Computing for Distributed Multimedia Systems*, Artech House Inc, ISBN 0-89006-884-4, pp. 146-148.

Lubachevsky, B. & Weiss, A. 1991, 'An Analysis of Rollback-Based Simulation', *ACM Transactions on Modeling and Computer Simulation*, vol. 1, no. 2, April 1991, pp. 154-193.

Macedonia, M. R., Zyda, M. J. 1995, 'A Taxonomy for Networked Virtual Environments', *Proceedings of the 1995 Workshop on Networked Realities*, Boston, MA, October 26-28, 1995. http://www-npsnet.cs.nps.navy.mil/npsnet/publications.html

Macedonia, M. R., Zyda, M. J., Pratt, D. R., Barham, P. T. & Zeswitz, S. 1994, 'NPSNET: A Network Software Architecture for Large Scale Virtual Environments', Presence, Vol. 3, No. 4. http://www-npsnet.cs.nps.navy.mil/npsnet/publications.html

Macedonia, M. R., Zyda, M. J., Pratt, D. R., Brutzman, D. P. & Barham, P. T. 1995, 'Exploiting Reality with Multicast Groups', *IEEE Computer Graphics and Applications*, September 1995, pp. 38-45.

Mandeville, J., Furness III, T., Kawahata, M., Campbell, D., Danset, P., Dahl, A., Dauner, J., Davidson, J., Howell, J., Kandie, K. & Schwartz, P. 1995, 'GreenSpace: Creating a distributed Virtual Environment for Global Applications', *Proceedings of the IEEE Networked Reality Workshop*, October 26-28, 1995, Boston, MA.

Miller, D. C. & Thorpe, J. A. 1995, 'SIMNET: The Advent of Simulator Networking', *Proceedings of the IEEE*, vol. 93, no. 8, August 1995.

Ng, Yu-Shen. 1997, 'Designing Fast-Action Games for the Internet', *Game Developer: Online Gaming*, Winter 1996/1997, Miller Freeman Inc, ISSN 1073-922X. http://www.gdmag.com/

Nicol, D. & Fujimoto, R. 1994, 'Parallel Simulation Today', *Annals of Operations Research*, pp. 249-286.

Nicol, D. M. 1991, 'Performance Bounds on Parallel Self-Initiating Discrete-Event Simulations', *ACM Transactions on Modeling and Computer Simulations*, vol. 1, no. 1, pp. 24-50.

Pratt, D. R. 1993, 'A Software Architecture for the Construction and Management of Real-Time Virtual Worlds', Department of Computer Science, Naval Postgraduate School, Monterey, California, Ph.D. Thesis, pp. 141-147.

Preiss, B. R., Loucks, W. M. & Macintyre, I. D., 1994, 'Effects of the Checkpoint Interval on Time and Space in Time Warp', *ACM Transactions on Modeling and Computer Simulation*, Vol. 4, No. 3, July 1994, pp. 223-253.

Pullen, R. & Garrett, R. 1995, 'Future Technology Challenges in Distributed Interactive Simulation', *IEEE Computer Graphics and Applications*, vol. 83, no. 8, August 1995.

Rahmat, O. 1997, 'In The News: The Latest News on the 3D Scene', *3D Design*, vol. 3, no. 4, April 1997, Miller Freeman Inc., ISSN 1083-5288, http://www.3d-design.com/

Reddy, J. M. & Wood, D. C. 1995, 'Networking Technology and DIS', *IEEE Computer Graphics and Applications*, Vol. 83, No. 8, August 1995.

Roberts, D. J., Sharkey, P. M. & Sandoz, P. D. 1995, 'A Real-time Predictive Architecture for Distributed Virtual Reality', *Proceeding of the Interactive Virtual Environments '95 Symposium*, Iowa, USA.

Ryan, M. D. & Sharkey, P. M. 1997, 'Use of Causal Volumes in Distributed Virtual Reality', IEEE International Conference on Systems, Man and Cybernetics, Orlando, Florida, 12-15 October, 1997.

Sandoz, P., Sharkey, P. & Warwick, K. 1994, 'Collision Prediction in Multi-User Artificial Worlds', *Proceedings of the IEEE 1994 Symposium on Computer Intensive Methods in Control and Signal Processing*, Prague, Czech Republic.

Sarin, S. & Greif, I. 1985, 'Computer-Based Real-Time Conferencing Systems', *IEEE Multimedia Communications*, 0018-9162/85/1000-003, October 1985.

Savetz, K., Randall, N., Lepage , Y. 1998, *MBONE: Multicasting Tomorrow's Internet*, ISBN: 1568847238

Shapiro, J. 1996, *Collaborative Computing: Multimedia Across the Network*, AP Professional, ISBN 0-12-638675-7.

Sharkey, P. M. & Kumar, A. 1995, 'Continuous Pre-warping of Time for Multi-user Interactive Virtual Environments', *Proceedings of the SPIE International Symposium on Telemanipulator & Telepresence Technologies II*, Philadelphia, USA, October 22-26 1995.

Sharkey, P. M., Roberts, P. D., Sandoz, R. & Cruse, R. 1996, 'A Single-user Perception Filter for Multi-user Distributed Virtual Environments', *Proceeding of the International Workshop on Collaborative Virtual Environments*, Nottingham, UK, June 1996.

Shiflett, J. E., Lunceford W. H. & Willis, R. P. 1995, 'Applications of Distributed Interactive Simulation Technology Within the Department of Defense', *IEEE Computer Graphics and Applications*, Vol. 83, No. 8, August 1995.

Sims, D. 1995, 'VR World 95: The New Element in VR Is People', *IEEE Computer Graphics and Applications*, 15: 4, July, 1995.

Singh, G., Serra, L., Png, W., Wong A., Ng, H. 1995, 'BrickNet: Sharing Object Behaviours on the Net', *VRAIS 95*.

Singhal, S. K. 1996, 'Effective Remote Modeling in Large-Scale Distributed Simulation and Visualization Environments', Ph.D. Thesis, Department of Computer Science, Stanford University.

Sommerville, I. 1992, *Software Engineering*, Fourth Edition, Addison-Wesley, ISBN 0-201-56529-3.

Steinman, J. S. & Wieland, F. 1994, 'Parallel Proximity Detection and the Distributed List Algorithm', *Proceedings of the 1994 workshop on Parallel and Distributed Simulation (PADS'94)*, pp. 3-11.

Steinman, J. S. 1995, 'Scalable Parallel and distributed Military Simulations Using the SPEEDES Framework', California Institute of Technology, Jet Propulsion Laboratory, Pasadena, California, ftp://pebbles.jpl.nasa.gov/pub/SPEEDES_Papers

Steinman, J. S., Lee, C. A., Wilson, L. F. & Nicol, D. M. 1995, 'Global Virtual Time and Distributed Synchronization', *Proceedings of the 1995 Workshop on Parallel and Distributed Simulation (PADS'95)*

Stytz, M.R. 1996, 'Distributed Virtual Environments', *IEEE Computer Graphics and Applications*, May 1996.

Tanenbaum, A. S. 1996, *Computer Networks*, 3rd Edition, Prentice Hall, ISBN 0-13-394248-1.

Trueman, P. 1995, 'Talking on the Internet', *UTAS*, September 1995, University of Tasmania.

Walters, R. C. 1996, 'Time Synchronization in Spline', Technical Report, Mitsubishi Electric Research Laboratories, Cambridge Research Center, Cambridge, Massachusetts, http://www.merl.com/

Walters, R. C., Anderson, D. B., Barrus, J. W., Brogan, D. C., Casey, M. A., McKeown, S. G., Nitta, T., Sterns, I. B., Yerazunis, W. S., 1996, 'Diamond Park and Spline: A Social Virtual Reality System with 3D Animation, Spoken Interaction, and Runtime Modifiability', Technical Report, Mitsubishi Electric Research Laboratories, Cambridge Research Center, Cambridge, Massachusetts, http://www.merl.com/

Wang, Q. 1994, 'Networked Virtual Reality', Master's Thesis, Department of Computing, University of Alberta, Edmonton, Alberta.

Wang, Q., Green, M. & Shaw, C. 1995, 'EM — An Environment Manager For Building Networked Virtual Environments', *Proceedings of the IEEE Virtual Reality Annual International Symposium (VRAIS)*, March 1995, IEEE Service Center, ISBN 0-8186-7084-3.

Whetten, B., Montgomery, T. & Kaplan, S. 1995, 'A High Performance Totally Ordered Multicast Protocol'. *Proceedings of the conference on Theory and Practice in Distributed Systems*, Springer Verlag, LCNS 938. ftp://research.ivv.nasa.gov/pub/doc/RMP/RMP_dagstuhl.ps

Woodson, W. E. 1987, *Human Factors Reference Guide for Electronics and Computer Professionals*, New York, McGraw-Hill.

Zeswitz, S. R. 1993, 'NPSNET: Integration of Distributed Interactive Simulation (DIS) Protocol for Communication Architecture and Information Interchange', Master's Thesis, Computer Science Department, Naval Postgraduate School, Monterey, CA 93943-5000.